MASTER EN SCIENCES INFORMATIQUES

Définition d'un sous-ensemble du langage Prolog.

Proposition d'une méthode de démonstration de programmes applicable à ce sous-ensemble

Derroitte, Marc

Award date:
1986

Awarding institution:
Universite de Namur

Définition d'un sous-ensemble du langage Prolog.

Proposition d'une méthode de démonstration de programmes applicable à ce sous-ensemble.

Marc Derroitte

Mémoire réalisé en vue
de l'obtention du titre
de Licencié et Maître
en Informatique

Promoteur : Baudouin Le Charlier

Namur

Septembre 1986

Remerciements.

Plus qu'un devoir, c'est un plaisir pour moi de remercier Monsieur Baudouin Le Charlier, promoteur de ce mémoire. Sa contribution à la réalisation de celui-ci est grande; ses conseils et ses idées me guidèrent d'un bout à l'autre de mon travail.

J'exprime toute ma gratitude à Monsieur Michel Vanden Bossche-Marquette et à tous les collaborateurs de la société B.I.M. à Everberg, pour leur accueil et leur aide durant mon stage.

Je présente enfin mes excuses et mes remerciements à tous ceux que j'ai dérangé durant la réalisation de ce mémoire, les uns par mes nombreuses questions, les autres par l'utilisation abusive de leur matériel.

Table des matières.

Introduction.

Ce chapitre présente les motivations de ce travail, propose un bref tour d'horizon de l'état actuel de la recherche en la matière, et définit nos hypothèses et objectifs principaux. Il se termine par une présentation succincte de la façon dont les chapitres suivants seront organisés.

1. LE PROBLEME.

Pour le non-initié, Prolog semble présenter bon nombre d'avantages décisifs par rapport à d'autres langages de programmation. Les notions de faits et de règles permettent de modéliser très facilement beaucoup de problèmes. Les paramètres d'une procédure peuvent être considérés indifféremment comme des données ou des résultats, ce qui permet une utilisation souple et multiple des procédures[1]. Et surtout, la présence du Tout-Puissant Moteur d'Inférence garantit un fonctionnement correct des programmes, sans aucune difficulté de programmation.

Malheureusement, tout n'est pas aussi simple : dès que l'on tente de tirer des renseignements intéressants de son petit arbre généalogique familial, on s'aperçoit que s'il est impossible de retrouver les noms de certains cousins, les noms de certains autres, par contre, sont répétés indéfiniment...

Plus tard, on se rend compte que la modélisation du réel pose pas mal de problèmes, que beaucoup de procédures ne sont pas réversibles, et qu'une bonne structuration des programmes ainsi qu'un contrôle explicite de leur exécution sont malgré tout nécessaires.

Face à tout celà, il nous semble que l'élaboration d'un cadre méthodologique qui permettrait d'augmenter la fiabilité des programmes construits avec Prolog serait d'une utilité appréciable. C'est le but ultime de ce travail.

---

1. Cette caractéristique est connue sous le nom de *réversibilité*.

2. ETAT ACTUEL DE LA QUESTION.

Classiquement, les recherches concernant l'approche "logique" de la programmation s'orientent selon deux grandes tendances bien établies.

La première s'attache à développer les connaissances et les théories relatives à la logique des prédicats, afin de construire un cadre conceptuel solide. On y retrouve des auteurs tels que R.A. Kowalski [KOWA 79].

La seconde, considérant la logique pure comme insuffisante en pratique, se concentre sur la conception de langages de programmation réellement opérationnels, dont l'implémentation soit la plus efficace possible, quelquefois en s'éloignant considérablement de cette logique. Les travaux de W.F. Clocksin et C.S. Mellish [CLO 84], ainsi que des langages tels que le Bim-Prolog [BIMP 85] s'inscrivent dans cette approche.

Il semble actuellement que naisse une approche résolument nouvelle, dont les tenants tentent de concilier la nécessité d'un cadre théorique rigoureux, et les exigences pratiques concernant l'efficacité des langages. Elle consiste à élaborer un certain nombre de méthodes permettant une utilisation efficace et fiable de ces langages. Les travaux de Y. Deville [DEVI 85] s'orientent en ce sens.

Dans la limite des hypothèses que nous allons maintenant préciser, ce mémoire se voudrait une contribution originale - du moins à notre connaissance - à cette troisième approche.

3. HYPOTHESES ET OBJECTIFS.

A la base, le but des langages tels que Prolog est de fournir un outil de haut niveau qui permette au programmeur de raisonner sur des problèmes, et de résoudre ceux-ci, à un niveau conceptuel élevé, sans devoir se préoccuper de tous les détails fastidieux concernant l'implémentation des solutions trouvées. La gestion de ces détails est alors prise en charge par le langage lui-même.

On peut sans doute d'ailleurs trouver là la cause de la croyance, actuellement assez répandue, selon laquelle un programme Prolog exprime en lui-même sa propre spécification.

Or, on remarque que l'utilisation de Prolog est difficile, et donne lieu à de nombreux programmes incorrects. Pour éviter cela, le programmeur qui espérait se contenter d'une sémantique déclarative est obligé d'introduire un contrôle explicite du fonctionnement de ses programmes, et de faire référence aux mécanismes de l'interpréteur lui-même, recourant ainsi à des notions relevant d'une sémantique purement procédurale.

Le moins qu'on puisse dire de cette situation est que le but initial n'est peut-être pas atteint.

Une solution possible pour résoudre le problème serait de considérer qu'on ne peut pas échapper à la dualité sémantique déclarative / sémantique procédurale. Elle consisterait à proposer une méthodologie permettant d'envisager successivement ces deux aspects, en élaborant d'abord la solution en termes déclaratifs, puis en introduisant les notions plus opérationnelles concernant le contrôle de l'exécution. Cette façon de faire, délimitant clairement les deux aspects, permettrait de construire des programmes plus fiables [DEVI 85].

Une autre solution partirait de l'idée que s'il est difficile d'utiliser Prolog, c'est parce que les mécanismes de base par lesquels le langage transforme une solution déclarative en une solution procédurale exécutable sont des mécanismes excessivement compliqués à appréhender. Les problèmes rencontrés par les programmeurs viendraient alors de ce qu'ils construisent leurs solutions en faisant constamment référence à des mécanismes qu'en fait, ils ne maîtrisent pas totalement.

Dans cette optique, la solution consisterait alors à proposer un modèle de ces mécanismes qui en fournirait une vue simplifiée et beaucoup plus maîtrisable. Une première version (limitée) d'un tel modèle pourrait se définir comme un sous-ensemble de Prolog, dont la caractéristique principale serait qu'il représente un langage déterministe.

En conséquence, pour un jeu de valeurs donné des paramètres d'une procédure, il n'y aurait au plus qu'une solution possible. Cette restriction permettrait de maîtriser le *backtracking* et ainsi de rendre inutile l'usage du *cut* comme moyen de contrôle supplémentaire.

Tout ceci doterait ce langage d'une sémantique opérationnelle beaucoup plus simple, basée principalement sur l'unification. Cela autoriserait, d'une part, la construction de raisonnements aisément maîtrisables, et d'autre part, l'usage de techniques de démonstration classiques telles que l'induction et la récursivité.

Pour écrire un programme, on pourrait alors raisonner en se limitant aux mécanismes propres à ce langage, et construire les programmes Prolog correspondants, ce qui serait toujours possible, puisque ce langage est un sous-ensemble de Prolog.

Il est clair que ce nouveau langage, déterministe, verra sa puissance limitée par rapport à Prolog. Cependant, en pratique, on peut constater que cette puissance n'est pas toujours exploitée complètement par les programmeurs. Nous montrerons dans la suite que ce langage est en fait suffisamment puissant pour exprimer la plupart des problèmes que nous avons rencontrés dans une application réalisée lors d'un stage précédant ce mémoire, et qu'en outre, un certain nombre d'autres problèmes qui ne pourraient pas être réduits tels quels à ce sous-ensemble, gagneraient en clarté et en fiabilité s'ils étaient repensés dans les termes de ce nouveau langage.

4. STRUCTURATION DES CHAPITRES SUIVANTS.

Dans une première partie, nous définirons complètement le langage dont il a été question ci-dessus, que nous baptiserons D.Log.[2]. Nous montrerons que ce langage est universel, et nous le comparerons avec un Prolog particulier, le Bim-Prolog.

Dans une seconde partie, sur base d'un cas pratique, nous présenterons une utilisation de D.Log., et nous proposerons une méthodologie de spécification et de validation, d'usage général, qui s'applique particulièrement bien à ce langage.

---

2. "D.Log." pour "Deterministic Prolog"

Partie 1.
Le langage D.Log.

1. DEFINITION DU LANGAGE.

1.1. Présentation intuitive.

Nous présentons ici une brêve description du langage, en expliquant simultanément les caractéristiques essentielles de sa syntaxe et de sa sémantique.

- Un *programme* est un ensemble de procédures.

- Une *procédure* définit des règles de calculs d'une suite de termes résultats $t'_1 , \dots , t'_n$ en fonction de termes données $t_1 , \dots , t_n$.

Les termes résultats sont toujours des particularisations des termes données.

- Une procédure se compose d'une suite de règles de calcul appelées *clauses* qui seront essayées successivement jusqu'à ce que l'une d'elles s'applique.

Lorsqu'une règle s'applique, on n'essaie pas les autres.

Pour cette raison, le langage est déterministe, au contraire de Prolog.

- Une *règle* se présente sous la forme :

$p(t_1 , \dots , t_n) :- S_1 , \dots , S_m .$

(avec n et m >= 0)

où ◇ p est appelé tête de la clause,

◇ les t_i sont des termes,

◇ '$S_1 , \dots , S_m$' est appelé suite de buts.

remarques :

si n = 0 , la règle se présente sous la forme $p :- S_1 , \dots , S_m$.

si m = 0 , la règle se présente sous la forme $p(t_1 , \dots , t_n)$.

- L'*exécution d'une règle* consiste en l'exécution de tous les buts S_i de sa suite de buts de telle sorte à obtenir les termes résultats t'_1 , ... , t'_n .

- Un *but* se présente sous la forme :

$p(par_1 , ... , par_n)$

(avec n >= 0)

où ◇ les par_i sont des termes.

- L'*exécution d'un but* consiste à appliquer une procédure à la suite de termes (par_1 , ... , par_n) .

- L'*application de la procédure p à la suite de termes (par_1 , ... , par_n)* consiste à déterminer si p est le nom d'un prédicat primitif du langage, ou une tête de règle du programme.

Si p est le nom d'un prédicat primitif, celui-ci est exécuté avec, comme arguments, les termes par_1 , ... , par_n .

Si p est une tête de règle du programme, on essaye successivement les règles de même tête jusqu'à en trouver une qui s'applique, et on exécute celle-ci avec les termes données par_1 , ... , par_n .

- Un *terme* est soit une constante,
soit une variable (qu'on désigne par un nom précédé du caractère '_'),
soit une liste de termes,
soit une tête avec des arguments (ou en-tête).

- Pour trouver une règle qui s'applique dans une procédure, on compare la liste des termes t_i de l'en-tête de cette règle avec la liste des paramètres par_i , et on regarde s'ils sont *unifiables* deux à deux, c'est-à-dire :

◇ si un terme est une constante, le terme correspondant est la même constante,

◇ si un terme est une variable, le terme correspondant est n'importe quoi,

◇ si un terme est une liste, le terme correspondant est aussi une liste dont les éléments respectifs sont unifiables deux à deux,

◇ si un terme est une en-tête, le terme correspondant est aussi une en-tête de même tête, et dont les arguments respectifs sont unifiables deux à deux.

Exemple : soit la procédure :

```
append( [] , _X , _X ) .
append( [ _W | _X ] , _Y , [ _W | _Z ] ) :-
        append( _X , _Y , _Z ) .
```

L'application de cette procédure avec la liste de termes données ([1,2,3] , [4,5] , _L) s'exécute de la façon suivante :

- D'abord, on compare les termes données avec les paramètres de la première clause : celle-ci ne convient pas car le premier terme donnée n'est pas la liste vide ([1 , 2 , 3] ≠ []),

- Par le même raisonnement, on se rend compte que la seconde règle peut être appliquée, on l'applique, ce qui donne à _W la valeur 1 , à _X la valeur [2 , 3] , et à _Y la valeur [4 , 5].

- Ensuite on exécute le but qui est dans cette seconde clause, ce qui reporte le problème par un appel récursif de la procédure avec les termes données ([2,3] , [4,5] , _Z)

- On recommence le cheminement effectué ci-dessus, ce qui, en supposant que la procédure est correcte, donnera à _Z la valeur [2 , 3 , 4 , 5] .

_ On applique, pour terminer, la valeur des variables de l'appel récursif aux termes de l'appel initial, ce qui nous donne :

_L = [_W | _Z] = [1 | [2 , 3 , 4 , 5]] = [1 , 2 , 3 , 4 , 5]

La liste des termes résultats est donc :

([1,2,3] , [4,5] , [1,2,3,4,5]) .

Et nous obtenons bien comme troisième terme le résultat attendu de la procédure : une liste qui est la concaténation des deux listes données au départ.

1.2. Syntaxe de D.Log.

1.2.1. Précisions concernant la présentation.

◇ '< >$^{1..\infty}$ ' signifie ' un nombre quelconque (>= 1) de < >'.

◇ Les accolades { } délimitent des commentaires destinés à faciliter la lecture de la grammaire.

1.2.2. Règles syntaxiques.

< **programme** > ::= < procédure >$^{1..\infty}$

< procédure > ::= < clause >$^{1..\infty}$

< clause > ::= < en-tête > :- < suite de buts > .
| < fait >

< **question** > ::= **?-** < suite de buts > .

< fait > ::= < en-tête > .

< suite de buts > ::= < but >
| < but > , < suite de buts >

< but > ::= < en-tête >
| < expression >

< en-tête > ::= < tête > **(** < suite de termes > **)**
| < tête >

< expression > ::= < terme >
| < opérateur > < expression >
| < expression > < opérateur > < expression >
| **(** < expression > **)**

< opérateur > ::= **is** | **+** | **-** | ***** | **/** | **mod** | ****** | **trunc** | **round**
| **=** | **<** | **<=** | **>** | **>=** | **<>**

< tête > ::= < atome >

< suite de termes > ::= < terme >
| < terme > , < suite de termes >

< terme > ::= < variable >
| < constante >
| < liste >
| < tête > (< suite de termes >)

< liste > ::= [< suite de termes >]
| [< suite de termes > | < liste >]

< constante > ::= < atome >
| ' < caractère >$^{1..\infty}$ '
| < entier >
| < réel >

< variable > ::= < underscore >< alphanumérique >$^{1..\infty}$
| < underscore >

< entier > ::= < nombre >
| -< nombre >

< réel > ::= < entier >.< nombre >
| < entier >.< nombre >< exposant >
| < entier >< exposant >

< exposant > ::= E < entier >
| e < entier >

< nombre > ::= < chiffre >$^{1..\infty}$

< atome > ::= < lettre >< alphanumérique >$^{0..\infty}$

< caractère > ::= < alphanumérique >
| < délimiteur >
| < caractère spécial >

< délimiteur > ::= { caractère d'espacement }
| { caractère de tabulation }
| { caractère de passage à la ligne }

< alphanumérique > ::= < lettre >
| < chiffre >
| < underscore >

< lettre > ::= **A | B | ... | Z | a | b | ... | z**

< chiffre > ::= **0 | 1 | ... | 9**

< underscore > ::= _

< caractère spécial > ::= { n'importe quel caractère imprimable différent d'une lettre, d'un chiffre, d'un délimiteur et de ' }

1.2.3. Remarques.

◇ **Cette syntaxe est directement inspirée de la syntaxe de Bim-Prolog.**

◇ **Des commentaires délimités par des accolades ( ) peuvent se trouver dans un programme écrit en D.Log. et sont destinés à documenter celui-ci.**

1.3. Sémantique de D.Log.

1.3.1. Concepts de base.

a. *Fichier*

Un fichier est caractérisé par :

- ◇ un **nom physique** et un **nom logique**,
- ◇ un **état** : - ouvert en lecture, écriture, 'ajout'
  - fermé (nom logique indéfini),
- ◇ un **contenu** : suite de caractères ASCII,
- ◇ un **contenu accessible** : suffixe du contenu (défini si ouvert en lecture).

b. *Variable de contexte*

Une variable de contexte est l'association

- ◇ d'un **nom de variable**,
- ◇ d'une **'case mémoire'** pouvant contenir la représentation d'un terme.

c. *Terme de base*

Un terme de base est un terme qui ne contient aucune variable.

On parlera également de 'terme complêtement instancié'.

d. *Base de faits*

Une base de faits est constituée d'un **ensemble de faits**.

remarque : sémantiquement, on pourra considérer la base de faits comme un 'fichier à accès direct' en mémoire centrale, contrairement aux fichiers disponibles qui sont à accès uniquement séquentiel.

e. *Substitution*

On définit une **substitution** $\mathcal{S}$ en fixant un ensemble de variables, et en associant à chacune un terme.

f. *Instanciation*

Une **instanciation d'un terme t par une substitution** S consiste à remplacer dans t chaque occurence d'une variable de S par la valeur qui lui est associée.

Si t est un **terme de base**, son instanciation par une quelconque substitution ne le modifie pas.

g. *Unification*

Soient t_1 et t_2 des termes.

On dit qu'ils sont **unifiables** s'il existe un terme t et une substitution S tels que t s'obtient aussi bien en instanciant t_1 par S qu'en instanciant t_2 par S.

On peut montrer que si t_1 et t_2 sont unifiables, il existe des termes t^* tels que tout terme t ci-dessus peut être obtenu par instanciation de t^*.

t^* est appelé **unificateur général** de t_1 et t_2.

Si t_1 et t_2 sont unifiables, on appellera **résultat de l'unification de t_1 et t_2**, une substitution S choisie arbitrairement telle qu'il existe un unificateur général t^* qui s'obtient en instanciant t_1 et t_2 par S.

h. *Matching*

t_1 'matche' avec t_2 (ou t_2 'matche' avec t_1) ssi t_1 et t_2 sont unifiables.

Les deux notions sont donc équivalentes.

1.3.2. Notion de contexte.

a. *Définition*

Un contexte est caractérisé par :

- ◇ un **programme**,
- ◇ un **ensemble de fichiers** - non fermés,
  - de noms logiques différents et de noms physiques différents,
- ◇ une **base de faits**,
- ◇ un ensemble de **variables de contexte** de noms différents.

b. *Opérations relatives aux contextes*

- On appelle **état d'un contexte** la donnée de
  - ◇ l'état de ses fichiers,
  - ◇ le contenu de sa base de faits,
  - ◇ les valeurs de ses variables.

- Opérations :

◇ **Création d'un contexte initial C .**

Un tel contexte sera obtenu en fixant :

- un programme,
- un ensemble vide de fichiers,
- une base de faits vide,
- un ensemble de variables de contexte ayant pour valeur leur nom.

◇ **Création d'un nouveau contexte C' dans un contexte C**

Un tel contexte sera défini en reprenant le programme, l'ensemble de fichiers et la base de faits du contexte C et en créant un nouvel ensemble de variables de contexte ayant toutes une valeur.

◇ **Instanciation d'un contexte C , par rapport à deux listes de termes (t_1, t_2, ..., t_n) et (t'_1, t'_2, ..., t'_n).**

La liste [t'_1, t'_2, ..., t'_n] doit être une instanciation de [t_1, t_2, ..., t_n].

Soit S , la substitution correspondante. On instancie les valeurs des variables de C par la substitution S .

◇ **Substitution $S(C)$ définie par un contexte C , à un instant donné.**

$S(C)$ est la substitution définie par les noms de variables de C et leur valeur.

◇ **Instanciation d'un terme t , dans un contexte C .**

Toutes les variables de t doivent être des noms de variables de C

On instancie t par la substitution $S(C)$

1.3.3. Sémantique proprement dite.

Cette sémantique est présentée sous la forme de trois *méta-algorithmes* de niveau conceptuel. Cette division permet de scinder le raisonnement nécessaire à la compréhension de l'exécution d'un programme. Ce raisonnement est ainsi fortement simplifié par rapport au raisonnement pour un programme en Prolog. En effet, en Prolog, il est difficile de prévoir l'effet de l'exécution d'une clause, sans tenir compte de tout le cheminement qui a précédé cette exécution.

A. **Algorithme ESB** (exécution d'une suite de buts)

◇ *données*: - une suite de buts S ,
- un contexte $\mathcal{C}$.

◇ *résultats*: - un booléen ' échec ' ,
- le contexte $\mathcal{C}$ éventuellement modifié.

On dit que l'exécution de la suite de buts S dans le contexte $\mathcal{C}$ échoue, si et seulement si l'exécution de l'algorithme se termine avec 'échec' = vrai ; elle réussit sinon.

◇ *algorithme*:

cas 1. **S est vide**,

L'algorithme produit pour résultat 'échec' = faux ,
le contexte $\mathcal{C}$ n'est pas modifié.

cas 2. **S = B S'**, où B est un but, et S' la suite de buts restante,

a. On exécute le but B dans le contexte $\mathcal{C}$: (algorithme EB(B , $\mathcal{C}$)),

b. Si cette exécution de EB échoue, celle de ESB(S , $\mathcal{C}$) se termine avec 'échec' = vrai et $\mathcal{C}$ dans l'état produit par EB(B , $\mathcal{C}$) ;
sinon, on exécute ESB(S' , $\mathcal{C}$) , avec $\mathcal{C}$ dans son nouvel état.

◇ *remarque*:

Une question ?- S. où S est une suite de buts

déclenche l'exécution de ESB(S , $\mathcal{C}$) où $\mathcal{C}$ est un contexte initial dont les noms des variables de contexte sont les noms des variables figurant dans S.(1)

B. **Algorithme EB** (exécution d'un but)

◇ *données*: - un but B ,

- un contexte $\mathcal{C}$.

◇ *résultat*: - le contexte $\mathcal{C}$ modifié.

◇ *algorithme*:

Soit p(t_1 , ... , t_n) la forme de B,

a. On instancie les termes t_1 , ... , t_n dans le contexte $\mathcal{C}$.

Notons t'_1 , ... , t'_n les termes obtenus.

b. On appelle la procédure p avec les termes (t'_1 , ... , t'_n) :

(algorithme AP(p , (t'_1 , ... , t'_n) , $\mathcal{C}$)).

b.1. Si cette exécution de AP échoue (renvoie 'échec' = vrai), l'exécution de EB(B , $\mathcal{C}$) échoue également.

b.2. Si cette exécution de AP réussit, elle produit pour résultat une liste de termes (t''_1 , ... , t''_n) qui sont des instanciations de t'_1 , ... , t'_n .

On instancie alors le contexte $\mathcal{C}$, par rapport aux listes de termes t'_1 , ... , t'_n et t''_1 , ... , t''_n .

---

1. exécution suivie de l'impression dans un fichier de sortie, des valeurs finales de ces variables.

C. **Algorithme AP** (application d'une procédure à une suite de termes)

◇ *données*: - un nom de procédure ou de primitive p ,

- une liste de termes (t_1 , ... , t_n) ,

- un contexte $\mathcal{C}$.

◇ *résultats*: - un booléen 'échec' ,

- si 'échec' = faux , une liste de termes (t'_1 , ... , t'_n) , instanciation de (t_1 , ... , t_n) .

- le contexte $\mathcal{C}$ modifié [2].

◇ *algorithme*:

cas 1. **p est un nom de procédure définie dans le programme P du contexte $\mathcal{C}$,**

Soit $p(s^1_1 , ... , s^1_n) :- S_1 .$

.

.

.

$p(s^m_1 , ... , s^m_n) :- S_m .$

la déclaration de p dans P ,

a. (i := 1) .

b. Soit i' la première valeur telle que (i <= i' <= m) et que les listes de termes ($s^{i'}_1$, ... , $s^{i'}_n$) et (t_1 , ... , t_n) puissent être unifiées.

Si cette valeur n'existe pas, l'exécution de AP(p , (t_1 , ... , t_n) , $\mathcal{C}$) échoue;

sinon on fait (i := i').

c. Les variables figurant dans la $i^{ème}$ clause de p sont renommées de telle sorte que cette clause n'ait pas de variable en commun avec (t_1 , ... , t_n) .

On note x^* le nouveau nom d'une variable x .

---

2. Les valeurs des variables de $\mathcal{C}$ ne seront pas modifiées.

d. On unifie l'en-tête de la clause renommée avec $p(t_1 , ... , t_n)$.

e. Soit $\mathcal{C}'$, le contexte créé dans $\mathcal{C}$, ayant pour ensemble X de variables, celles qui figurent dans la $i^{ème}$ clause de p et tel que :

$\forall x \in X$, la valeur de x est soit celle donnée à x^* par l'unification ci-dessus, soit x^* lui-même, si x ne figurait pas dans l'en-tête (dans ce cas on suppose que le résultat de l'unification n'utilise pas x^*) .

On exécute $ESB(S_i , \mathcal{C}')$.

f. Si cette exécution échoue, on fait (i := i + 1), puis on va en b.

Sinon l'exécution de $AP(p , (t_1 , ... , t_n) , \mathcal{C})$ réussit et renvoie pour résultat la liste des termes $(t'_1 , ... , t'_n)$ résultant de l'instanciation de $(s^i_1 , ... , s^i_n)$, dans $\mathcal{C}'$ (modifié par l'exécution de $ESB(S_i , \mathcal{C}')$).

cas 2. **p est un nom de prédicat primitif** (3),

L'exécution de $AP(p , (t_1 , ... , t_n) , \mathcal{C})$ consiste en l'appel du prédicat primitif p .

Cet appel réussit ou échoue en fonction des termes $t_1 , ... , t_n$ et du contexte $\mathcal{C}$, et modifie éventuellement ceux-ci (voir la définition de ces prédicats ci-dessous).

1.3.4. Prédicats primitifs du langage.

Les prédicats primitifs sont définis par une précondition et un effet.

La **précondition** détermine les conditions sur les arguments $(t_1 , ... , t_n)$ du prédicat pour que celui-ci réussisse.

Sinon l'exécution de $AP(p , (t_1 , ... , t_n) , \mathcal{C})$ échoue.

Dans le cas où la précondition est vérifiée, l'**effet** détermine la liste de termes $t'_1 , ... , t'_n$, instanciation de $t_1 , ... , t_n$, et la modification du contexte $\mathcal{C}$.

Lorsque les valeurs $t'_1 , ... , t'_n$ ne sont pas explicitement définies, elles sont égales à $t_1 , ... , t_n$.

---

3. ou primitive.

A. Primitives d'entrée/sortie

◇ **fopen**

précondition : - n = 3 ,

- t_1 , t_2 sont des noms de fichiers,

- t_3 est une constante égale à 'w' , 'r' ou 'a'.

effet : le fichier de nom physique t_2 est ouvert en lecture (t_3 = 'r'), en écriture (t_3 = 'w') ou en 'ajout' (t_3 = 'a') et est ajouté au contexte courant avec le nom logique t_1 .

◇ **fclose**

précondition : - n = 1 ,

- t_1 est le nom logique d'un fichier appartenant au contexte courant.

effet : le fichier de nom logique t_1 est fermé et est retiré du contexte courant.

◇ **eof**

précondition : - n = 1 ,

- t_1 est le nom logique d'un fichier du contexte, ouvert en lecture, dont le contenu accessible est vide[4] (donc, si le contenu accessible n'est pas vide, l'appel échoue).

◇ **nl**

précondition : - n = 1 ou 0,

- t_1 est le nom logique d'un fichier appartenant au contexte courant et ouvert en écriture ou en 'ajout' .

effet : un caractère de fin de ligne est ajouté à la suite du contenu du fichier de nom logique t_1 ou de nom logique 'stdout' si n était égal à 0 .

◇ **read**

précondition : - n = 2 ,

---

4. En Bim-Prolog, le contenu accessible vide ne suffit pas, il faut avoir lu le caractère de fin de fichier, ce qui se traduit par un *'read'* qui a échoué.

- t_1 est le nom logique d'un fichier du contexte, ouvert en lecture, dont le contenu accessible n'est pas vide et commence par un terme(5) ,

- t_2 est une variable.

effet : la variable t_2 est instanciée avec le premier terme du contenu accessible du fichier de nom logique t_1 , et ce terme est retiré du contenu accessible ainsi que le caractère '.' et le caractère de fin de ligne.

◇ **get0**

précondition : - n = 2 ,

- t_1 est le nom logique d'un fichier du contexte, ouvert en lecture, dont le contenu accessible n'est pas vide ,

- t_2 est une variable.

effet : t'_2 est le code ASCII du premier caractère du contenu accessible du fichier de nom logique t_1 , et ce caractère est retiré du contenu accessible.

◇ **write**

précondition : - n = 2 ou 1 ,

- t_1 est le nom logique d'un fichier du contexte, ouvert en écriture ou en 'ajout', (si n = 2)

- t_n est un terme.

effet : le terme t_n est ajouté à la suite du fichier de nom logique t_1 ou de nom logique 'stdout' si n était égal à 1 , avec la transformation suivante : si le terme contenait des variables leur nom est remplacé par un caractère '_' suivi d'un nombre, et si le terme était une suite de caractères différente d'un atome, ce terme est mis entre 'quotes' (6).

◇ **put**

précondition : - n = 2 ou 1 ,

- t_1 est le nom logique d'un fichier du contexte, ouvert en écriture ou en 'ajout', (si n = 2)

- t_n est un entier.

---

5. c'est-à-dire une suite de caractères correspondant à la syntaxe d'un terme et suivie du caractère '.' et du caractère de fin de ligne.

6. voir définition de writeq dans le manuel de Bim-Prolog, page 4 - 2.

effet : le caractère qui a le code ASCII égal à t_n modulo 128 est ajouté à la suite du fichier de nom logique t_1 ou de nom logique 'stdout' si n était égal à 1 .

Remarque :

Les primitives citées ci-dessus constituent un sous-ensemble minimum des primitives utiles à la manipulation de fichiers. On pourra éventuellement en rajouter d'autres telles que *tell*, *see*, *told*, *seen*, *tell_err*, *told_err*, *spaces*, *skip*, *display*, ... Leur définition devrait correspondre à celle qui est donnée dans le manuel Bim-Prolog.

B. Primitives d'interaction avec la base de faits

◇ **consult**

précondition : - n = 1 ,

- t_1 est le nom physique d'un fichier du contexte, le fichier doit être fermé et contenir la représentation sous forme de chaînes de caractères d'un ensemble de faits. De plus, les faits doivent être regroupés selon leur tête [7].

effet : ajouter cet ensemble de faits à la base de faits.

◇ **assert**

précondition : - n = 1 ,

- t_1 est un terme .

effet : t_1 est ajouté à la base de faits.

◇ **retract**

précondition : - n = 1 ,

- t_1 est instanciable avec un fait qui se trouve dans la base de faits [8] .

effet : un fait avec lequel t_1 est instanciable, est retiré de la base de faits.

◇ **retractall**

précondition : - n = 1 ,

---

7. c'est-à-dire : les faits ayant même tête doivent se suivre dans le fichier.

8. c'est différent de Bim-Prolog, voir page 5-4 du manuel.

- t_1 est un atome qui est la tête d'au moins un fait qui se trouve dans la base de faits.

effet : tous les faits dont la tête est t_1 sont retirés de la base de faits.

◇ **abolish**

précondition : - n = 2 ,

- t_1 est un atome,

- t_2 est un entier,

- il existe dans la base de faits au moins un fait dont la tête est l'atome t_1 et dont le nombre de paramètres est l'entier t_2 .

effet : tous les faits dont la tête est t_1 et dont le nombre de paramètres est t_2 sont retirés de la base de faits.

◇ **foreachfact**

précondition : - n =2 ,

- t_1 est un terme,

- t_2 est un but.

effet : pour chaque fait 'fact' de la base de faits, unifiable avec t_1 ,

- on crée un nouveau contexte C' dans le contexte C de l'appel, avec, comme variables de contexte, les variables figurant dans t_1 , et comme valeurs de ces variables, les valeurs résultant de l'unification de t_1 avec fact;

- on exécute t_2 dans le contexte C' .

◇ **isfact**

précondition : - n = 1 ,

- t_1 est un terme unifiable avec un fait de la base de faits.

effet : t'_1 est un unificateur général de t_1 et d'un fait de la base de faits.

Remarque :

On pourra également rajouter d'autres primitives d'interaction avec la base de faits, si nécessaire, par exemple *bagof* et *setof* qui sont définies dans le manuel Bim-Prolog.

C. Primitives de tests

◇ **foreachmember**

précondition : - n = 3 ,

- t_1 est un terme,

- t_2 est une liste de termes,

- t_3 est un but.

effet : pour chaque terme 'el' de t_2 , unifiable avec t_1 ,

- on crée un nouveau contexte C' dans le contexte C de l'appel, avec, comme variables de contexte, les variables figurant dans t_1 , et comme valeurs de ces variables, les valeurs résultant de l'unification de t_1 avec el;

- on exécute t_3 dans le contexte C'.

◇ **ground**

précondition : - n = 1 ,

- t_1 est un terme complètement instancié, ou terme de base.

◇ **var**

précondition : - n = 1 ,

- t_1 est une variable.

◇ **atom , integer , real**

précondition : - n = 1 ,

- t_1 est respectivement un atome, un entier ou un réel.

◇ **opérateurs de comparaison** = , < , > , <> , <= , >=

précondition : - n = 2 ,

- t_1 et t_2 sont des entiers, des réels, des atomes,

- $t_1 = t_2$ ou $t_1 < t_2$ ou $t_1 > t_2$ ou $t_1 <> t_2$ ou $t_1 <= t_2$ ou $t_1 >= t_2$ (selon l'opérateur).

◇ **not**

précondition : - n = 1 ,

- t_1 est un but s'écrivant p(s_1 , ... , s_m) ,

effet : On exécute AP(p , (s_1 , ... , s_m) , $\mathcal{C}$).

- Si cette exécution échoue, celle de AP(not , t_1 , $\mathcal{C}$) réussit (avec $t'_1 = t_1$),
- Si elle réussit, celle de AP(not , t_1 , $\mathcal{C}$) échoue .

◇ **fail**

effet : L'exécution de AP(fail , () , $\mathcal{C}$) échoue toujours.

D. Primitives de conversion

◇ **ascii**

précondition : - n = 2 ,

- t_1 est une variable ou un caractère entre apostrophes,
- t_2 est une variable ou un entier,
- t_1 et t_2 ne sont pas tous deux des variables,
- si t_1 et t_2 sont des constantes, t_2 est le code ASCII de t_1.

effet : renvoie une liste (t'_1 , t'_2) de constantes telle que :

- t'_2 est le code ASCII de t'_1 ,
- t'_1 est une instanciation de t_1 ,
- t'_2 est une instanciation de t_2 ,

◇ =..

précondition : - n = 2 ,

- t_1 est une variable ou une liste de termes dont le premier est un atome,
- t_2 est une variable ou un terme,
- t_1 et t_2 ne sont pas tous deux des variables,
- si t_1 et t_2 ne sont pas des variables, t_1 = [p , t'_1 , ... , t'_m] et t_2 = p(t'_1 , ... , t'_m).

effet : renvoie une liste (t1* , t2*) de termes telle que :

- t_1^* est une instanciation de t_1 ,
- t_2^* est une instanciation de t_2 ,
- $t_1^* = [\ p\ ,\ t'_1\ ,\ \ldots\ ,\ t'_m\ ]$ et $t_2^* = p(\ t'_1\ ,\ \ldots\ ,\ t'_m\ )$.

◇ **atomtolist**

précondition :
- n = 2 ,
- t_1 est une variable ou un atome,
- t_2 est une variable ou une liste de caractères entre apostrophes,
- t_1 et t_2 ne sont pas tous deux des variables,
- si t_1 et t_2 ne sont pas des variables, t_1 s'écrit $c_1c_2..c_m$ et $t_2 = [\ 'c_1'\ ,\ 'c_2'\ ,\ \ldots\ ,\ 'c_m'\ ]$.

effet : renvoie une liste (t_1^* , t_2^*) de termes telle que :

- t_1^* est une instanciation de t_1 ,
- t_2^* est une instanciation de t_2 ,
- t_1^* s'écrit $c_1c_2...c_m$ et $t_2^* = [\ 'c_1'\ ,\ 'c_2'\ ,\ \ldots\ ,\ 'c_m'\ ]$.

◇ **name**

Idem que atomtolist, sauf que pour t_2 , on a une liste d'entiers au lieu d'une liste de caractères entre apostrophes, et

si t_1 s'écrit $c_1c_2...c_m$, alors $t_2 = [\ a_1\ ,\ a_2\ ,\ \ldots\ ,\ a_m\ ]$ avec a_i = code ASCII de c_i.

◇ **arg**

précondition :
- n = 3 ,
- t_1 est un entier,
- t_2 est un terme,
- t_1 est positif et inférieur ou égal au nombre d'arguments du terme t_2 ,
- t_3 est une variable.

effet : t'_3 est le $t_1^{ème}$ argument du terme t_2 .

E. Primitive d'évaluation d'expressions

◇ **is**

précondition : - n = 2 ,

- t_1 est un entier, un réel, un atome ou une variable.
- t_2 est une expression ne contenant pas de variable.
- si t_1 n'est pas une variable t_1 est la valeur de t_2.

effet : renvoie (t'_1 , t'_2) où t'_1 est la valeur de t'_2 et $t'_2 = t_2$.

Les expressions sont construites avec les opérateurs binaires : '+' (addition), '-' (soustraction), '*' (multiplication), '/' (division), 'mod' (modulo : reste de la division par), '**' (exposant) et les opérateurs unaires : 'trunc' (arrondit à la partie qui précède la virgule), 'round' (arrondit à l'entier le plus proche), '+' (plus), '-' (moins).

Elles sont interprétées en utilisant la règle d'association à gauche et les priorités suivantes pour les opérateurs : '**' > opérateurs unaires '+', '-', 'trunc', 'round' > 'mod', '/', '*' > '+', '-'.

2. UTILISATION DU LANGAGE.

2.1. Universalité de D.Log.

Nous allons démontrer l'universalité du langage que nous venons de définir par une **traduction du LISP pur en D.Log.** Etant donné que le LISP est un langage universel, cette démonstration permettra d'affirmer que le D.Log est également un langage universel.

Concrètement, cela signifie que l'on pourra 'tout' programmer en D.Log, et que n'importe quel programme écrit en un langage quelconque pourra être transformé en un programme équivalent écrit en D.Log.

Donc, en particulier, un programme écrit en Prolog pourra être traduit en D.Log et il sera possible de trouver un équivalent à toutes les notions ou constructions utilisées en Prolog.

Démonstration :

A. Les éléments de base du LISP sont les suivants :

◇ les **fonctions primitives** *cons* , *eq* , *car* , *cdr* , *atom* ,

◇ les expressions : - **atomes** et **listes** (ne contenant pas des variables),

- **variables** ,

- if bexpr then $expr_1$ else $expr_2$

(expressions conditionnelles)

- f($expr_1$, ... , $expr_n$)

(applications de fonctions)

◇ les **déclarations** : f(x_1 , ... , x_n) = expr

Un **programme** est un ensemble de déclarations.

exemple :

```
append( x , y ) =
        if eq( x , nil )
        then y
        else cons( car( x ) , append( cdr( x ) , y ) )
```

B. Langage intermédiaire.

Pour faciliter la traduction du LISP en D.Log., on introduit un langage intermédiaire (sous-ensemble du LISP) équivalent au LISP en puissance, mais d'une syntaxe plus rudimentaire.

1. Les éléments de base de ce langage sont les suivants :

◇ les **fonctions primitives du LISP** ,

◇ les expressions suivantes :

- **constantes** (atomes et listes),
- **variables** ,
- les **expressions conditionnelles** de la forme ,

$$\underline{\text{if}}\ f(x_1, \ldots, x_n)\ \underline{\text{then}}\ g(x_1, \ldots, x_n)\ \underline{\text{else}}\ h(x_1, \ldots, x_n)$$

où les x_i sont des variables,

et f , g , h des noms de fonctions,

- les **applications de fonctions** de la forme ,

$$f(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n))$$

où les x_i sont des variables,

et $f, h_1, \ldots, h_m$ des noms de fonctions,

◇ les **déclarations** de fonctions de la forme :

$$f(x_1, \ldots, x_n) = \text{expr}$$

où expr est une expression comme ci-dessus.

2. On peut **traduire** un programme LISP en un programme du langage intermédiaire en appliquant les règles suivantes :

Soient $x_1, \ldots, x_n$, les variables figurant dans le programme LISP.

◇ A toute expression expr , on associe une déclaration de fonction du langage intermédiaire, calculant la même valeur, comme suit :

+ Si expr est une constante ou une variable :

$$f(x_1, \ldots, x_n) = \text{expr}$$

+ Si expr est de la forme :

  if $expr_1$ then $expr_2$ else $expr_3$

  sa traduction sera :

$$f(x_1, \ldots, x_n) = \underline{\text{if}}\ g_1(x_1, \ldots, x_n)\ \underline{\text{then}}\ g_2(x_1, \ldots, x_n)\ \underline{\text{else}}\ g_3(x_1, \ldots, x_n)$$

  où g_1 , g_2 , g_3 sont les noms des traductions de $expr_1$, $expr_2$, $expr_3$.

+ Si expr est de la forme :

  $g(expr_1, \ldots, expr_m)$

  sa traduction sera :

$$f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n))$$

  où h_1 , ... , h_m sont les noms des traductions de $expr_1$, ... , $expr_m$.

◇ A toute déclaration de fonction

$$f(y_1, \ldots, y_m) = expr$$

on associe la déclaration :

$$f(x_1, \ldots, x_n) = g(x_1, \ldots, x_n)$$

où g est la traduction de expr.

(la nouvelle fonction peut contenir plus de variables que l'ancienne, mais sa valeur ne dépend pas de ces autres variables.)

3. Moyennant la remarque ci-dessus, on vérifie facilement que les nouvelles fonctions calculent les mêmes valeurs que les anciennes.

4. **Exemple** : traduction de append dans le langage intermédiaire :

Il y a deux variables dans append , donc nous aurons deux paramètres dans toutes les définitions de fonctions :

fnil(x , y) = nil

fx(x , y) = x

fy(x , y) = y

$feq_1(x , y) = eq(fx(x , y) , fnil(x , y))$

$fcar_1(x , y) = car(fx(x , y))$

$fcdr_1(x , y) = cdr(fx(x , y))$

$fappend_1(x , y) = append(fcdr_1(x , y) , fy(x , y))$

$fcons_1(x , y) = cons(fcar_1(x , y) , fappend_1(x , y))$

$fsi_1(x , y) =$ if $feq_1(x , y)$ then $fy(x , y)$ else $fcons_1(x , y)$

Et la déclaration :

$append(x , y) = fsi_1(x , y)$

C. Traduction du langage intermédiaire en D.Log.

A chaque programme du langage intermédiaire, on associe un programme D.Log. équivalent dans le sens suivant :

A toute fonction f du programme de départ, à n arguments, on associe une procédure D.Log. pf à n+1 arguments telle que : pour toutes constantes $d_1 , \ldots , d_n , c$ avec $c = f(d_1 , \ldots , d_n)$, l'appel de la procédure pf avec les termes $(d_1 , \ldots , d_n , \_X)$ produit pour résultat les termes $(d_1 , \ldots , d_n , c)$.

1. **Règles de traduction** :

◇ $f(x_1 , \ldots , x_n) = t$ où t est une constante ou une variable devient :

$pf(\_X_1 , \ldots , \_X_n , t)$

◇ $f(x_1 , \ldots , x_n) = g(h_1(x_1 , \ldots , x_n) , \ldots , h_m(x_1 , \ldots , x_n))$

devient :

$pf(\_X_1 , \ldots , \_X_n , \_Z) :-$

$ph_1(\_X_1 , \ldots , \_X_n , \_Y_1) ,$

.

.

.

$ph_m(\_X_1 , \ldots , \_X_n , \_Y_m) ,$

$pg(\_Y_1 , \ldots , \_Y_m , \_Z) .$

◇ f(x_1 , ... , x_n) = <u>if</u> g_1(x_1 , ... , x_n)
<u>then</u> g_2(x_1 , ... , x_n)
<u>else</u> g_3(x_1 , ... , x_n)

devient :

```
pf( _X1 , ... , _Xn , _Z ) :-
        pg1( _X1 , ... , _Xn , _Y ) ,
        paux( _Y , _X1 , ... , _Xn , _Z ) .

paux( T , _X1 , ... , _Xn , _Z ) :-
        pg2( _X1 , ... , _Xn , _Z ) .
paux( _ , _X1 , ... , _Xn , _Z ) :-
        pg3( _X1 , ... , _Xn , _Z ) .
```

(en supposant que T veut dire 'vrai' et que tout le reste veut dire 'faux')

2. Primitives :

◇ <u>atom</u>

```
patom( _X , T ) :- atom( _X ) .
patom( _X , F ) .
```

◇ <u>eq</u>

```
peq( _X , _X , T ) .
peq( _ , _ , F ) .
```

◇ <u>car</u>

```
pcar( [ _X | _Y ] , _X ) .
```

◇ <u>cdr</u>

```
pcdr( [ _X | _Y ] , _Y ) .
```

◇ <u>cons</u>

```
pcons( _X , _Y , [ _X | _Y ] ) .
```

3. **Exemple** : **traduction de** append **en D.Log., à partir des fonctions définies dans le langage intermédiaire :**

```
pfnil( _X , _Y , nil ) .
pfx( _X , _Y , _X ) .
pfy( _X , _Y , _Y ) .
pfeq1( _X , _Y , _Z ) :-
        pfx( _X , _Y , _Z1 ) ,
        pfnil( _X , _Y , _Z2 ) ,
        peq( _Z1 , _Z2 , _Z ) .
pfcar1( _X , _Y , _Z ) :-
        pfx( _X , _Y , _Z1 ) ,
        pcar( _Z1 , _Z ) .
pfcdr1( _X , _Y , _Z ) :-
        pfx( _X , _Y , _Z1 ) ,
        pcdr( _Z1 , _Z ) .
pfappend1( _X , _Y , _Z ) :-
        pfcdr1( _X , _Y , _Z1 ) ,
        pfy( _X , _Y , _Z2 ) ,
        pappend( _Z1 , _Z2 , _Z ) .
pfcons1( _X , _Y , _Z ) :-
        pfcar1( _X , _Y , _Z1 ) ,
        pfappend1( _X , _Y , _Z2 ) ,
        pcons( _Z1 , _Z2 , _Z ) .
pfsi1( _X , _Y , _Z ) :-
        pfeq1( _X , _Y , _Z1 ) ,
        paux1( _Z1 ,_X , _Y , _Z ) .

paux1( T , _X , _Y , _Z ) :-
        pfeq1( _X , _Y , _Z ) .
paux1( _ , _X , _Y , _Z ) :-
        pfcons1( _X , _Y , _Z ) .
```

```
pappend( _X , _Y , _Z ) :-
        psi1( _X , _Y , _Z ) .
```

Tous les programmes écrits en LISP pur pourront donc être transformés, de la même manière, en programmes équivalents écrits en D.Log.

Par conséquent, D.Log. est bien un langage universel.

Remarques :

a. On constate que les programmes traduits du LISP ont la particularité d'être formés de règles qui ont un nombre quelconque de paramètres données suivis d'**un seul** paramètre résultat [1].

b. Signalons que cette démonstration n'a pas pour but de présenter une quelconque méthode de programmation. Elle a seulement un intéret théorique.

Pour s'en convaincre, il suffit de comparer la longueur du programme D.Log. de la fonction append présenté ci-dessus, avec le nombre de clauses (deux) de la même fonction présentée dans le paragraphe 1.1. de cette partie.

c. On constate également que le fait d'inclure dans le langage la possibilité de construire des procédures qui puissent échouer[2], n'est **théoriquement pas nécessaire**. En effet, dans la démonstration, on traduit une procédure LISP uniquement au moyen de procédures qui réussissent toujours.

Mais nous ne nierons pas l'utilité de procédures qui peuvent échouer, ni de procédures qui réussissent toujours mais dont certaines clauses peuvent échouer[3].

---

1. Ces notions seront définies de manière précise au point 1.1.1. de la seconde partie.

2. c'est-à-dire des procédures contenant une ou plusieurs clauses dont l'échec constitue un résultat souhaité dans certains cas et prévu dans la postcondition.

3. Une clause qui peut échouer est une clause dont le *matching* , pour un jeu de valeurs donné, peut être remis en cause par l'échec d'un de ses buts. Une procédure qui réussit toujours peut contenir une telle clause : il suffit en effet que cette clause soit suivie d'une autre, qui puisse *matcher* et réussir pour le même jeu de valeurs.

ce mécanisme permet en effet de faciliter la programmation et de raccourcir les programmes.

2.2. Commentaires sur la définition du langage.

Dans le domaine de la définition des langages, une analyse de la littérature nous permet d'affirmer que dans de nombreux cas, les langages sont définis de manière incomplète ou ambiguë. Nous avons donc particulièrement veillé, dans le cadre de cette définition de D.Log., à éviter ce travers.

On peut ainsi attirer l'attention sur les points suivants :

a. La syntaxe de D.Log. est entièrement définie. De plus, cette syntaxe et la présentation intuitive qui la précède permettent d'obtenir un niveau de compréhension satisfaisant, même sans connaissances préalables du Prolog.

b. Les mécanismes de base du langage Prolog (l'unification , le *backtracking* , le *cut*) sont, à notre avis, très complexes. Ce qui explique peut-être l'imprécision avec laquelle ils sont fréquemment définis.

Notre but étant de construire des raisonnements sur base de D.Log., il était indispensable, pour que le langage ait un intérêt pratique, de réduire cette complexité. Nous avons donc cherché un compromis entre le sous-ensemble de Prolog qui serait pris en compte et une simplicité de la sémantique.

Les trois méta-algorithmes qui définissent cette sémantique sont, à notre avis, clairs, précis et non ambigus. Ils utilisent au maximum des mécanismes simples qui permettent de reproduire l'effet global des mécanismes de base de Prolog et peuvent s'y substituer pour une large classe d'applications.

c. En ce qui concerne les prédicats primitifs, on se rend compte une fois de plus que les spécifications qui en sont données dans certains manuels sont parfois ambiguës ou ne prévoient pas tous les cas d'utilisation.

Ainsi, dans le cas du Bim-Prolog, qu'il nous a été donné d'utiliser, nous avons quelquefois dû tester le fonctionnement de prédicats dans un programme afin de nous

rendre compte de leurs effets, avant de pouvoir les utiliser effectivement (en espérant rester toujours dans les conditions 'normales' d'utilisation).

Les démonstrations de programmes étant impossibles dans ce contexte, nous avons eu recours à un mode de spécification qui, pensons-nous, permet de spécifier les prédicats de manière concise, et de lever les ambiguités possibles quant à leur exécution.

2.3. Conversions Bim-Prolog - D.Log.

2.3.1. Pour convertir un programme D.Log en Bim-Prolog.

D.Log. étant un sous-ensemble de Prolog construit sur base de Bim-Prolog, aucune modification n'est nécessaire pour transformer un programme de l'un dans l'autre, excepté la suivante.

Etant donné que D.Log. est déterministe, pour que le programme s'exécute de la même façon, il est nécessaire, en Bim-Prolog, d'ajouter un cut (!) comme dernier but de chaque clause. De cette manière, le *backtracking* ne pouvant pas fonctionner, il n'y aura jamais dans une procédure qu'une seule clause qui s'exécutera entièrement.

2.3.2. Pour convertir un programme Bim-Prolog en D.Log.

La démonstration d'universalité a prouvé que cette conversion était possible. Le programme présenté à titre d'illustration dans la seconde partie de ce mémoire a d'ailleurs pu être converti sans que cela pose de problème particulier.

En fait, s'il est vrai que beaucoup de procédures ne nécessitent que peu ou pas de modifications (celles qui sont déterministes), certaines, cependant, doivent être complètement repensées, parce qu'elles utilisent des mécanismes qui n'existent pas en D.Log.

Ainsi, pour ce qui est du *backtracking*, la plupart des procédures Prolog qui l'utilisent le font en réalité pour effectuer un traitement itératif, et renvoyer successivement plusieurs résultats. L'absence de *backtracking* en D.Log. oblige à rendre explicite ce caractère itératif, par exemple en définissant ces procédures de façon récursive ou en

utilisant des structures de données telles que la liste et la base de faits, pour lesquelles des prédicats de manipulation ont été ajoutés[4] afin de compenser l'absence de *backtracking*.

Du point de vue de la relation entre les procédures et la base de faits, il faut rappeler qu'en Prolog, les faits et les procédures ne sont pas différenciables, alors qu'en D.Log., la base de faits est indépendante. Dès lors, le test de la présence d'un fait, qui normalement se traduirait par l'appel d'une procédure, correspond en D.Log. à l'utilisation du prédicat primitif isfact. Par exemple, isfact(toto(_X)) est l'équivalent D.Log. de l'appel toto(_X) en Bim-Prolog.

Pour ce qui est du jeu de primitives utilisé, on peut noter que rien n'empêche d'en définir de nouvelles pour augmenter la puissance 'pratique' du langage. Ainsi, on pourra reprendre des prédicats de Bim-Prolog qui n'auraient pas été redéfinis en D.Log.

Pour terminer, on prendra en compte les remarques suivantes :

- Il est toujours possible de transformer une procédure pour qu'elle réussisse toujours en lui ajoutant en dernier lieu une clause supplémentaire capable de *matcher* pour toute valeur des paramètres. Cette clause pourrait en outre mettre à jour un paramètre booléen que l'on ajouterait à la liste des arguments de la procédure, afin de signaler les cas d'échec de la procédure initiale.

- Puisque les procédures D.Log. ne peuvent réussir qu'une fois, les séquences ' ! , fail ' peuvent être traduites par un 'fail' simple, moyennant une adaptation éventuelle de la procédure empêchant les clauses suivantes de réussir.

- Le ';' séparant deux buts alternatifs d'une même clause n'existe pas en D.Log. On pourra cependant réécrire la clause sous forme de deux clauses, ne comportant chacune qu'un des buts en question.

  Par exemple, la clause :

  ```
  alphanum( _X ) :- ( lettre( _X ) ; chiffre( _X ) ) .
  ```

  peut s'écrire :

  ```
  alphanum( _X ) :- lettre( _X ) .
  alphanum( _X ) :- chiffre( _X ) .
  ```

---

4. cfr la définition d'isfact, foreachfact et foreachmember.

Partie 2.
Justification des programmes.

1. METHODOLOGIE PROPOSEE.

Après avoir défini un langage, nous allons présenter une méthodologie applicable sur base de ce langage. Le but de cette méthodologie est de proposer un schéma de raisonnement simple et pratique qui permet de démontrer la correction de procédures Prolog.

La méthodologie comportera :

- un schéma de spécification,
- un schéma de démonstration,
- un schéma de documentation.

La méthode proposée est la suivante :

Toutes les procédures devraient être **spécifiées** selon le schéma de spécification, avec des adaptations éventuelles à ce schéma pour l'une ou l'autre procédure qui ne ferait pas exactement partie du sous-ensemble de Prolog qu'est D.Log.

De plus **les procédures d'une certaine complexité** devraient être **documentées** selon le schéma de documentation proposé.

Et **les procédures les plus complexes** devraient être explicitement **démontrées** selon le schéma de démonstration proposé.

Cette méthodologie s'applique à toutes les procédures qui entrent dans le cadre de la définition de D.Log.

◇ Dans le cas où l'on construit un programme Prolog, on veillera à construire les procédures, dans la limite du possible, en respectant les contraintes inhérentes au langage D.Log.

◇ Dans le cas où l'on démontre un programme déjà construit, on adaptera soit les procédures, soit les raisonnements.

Remarque : dans un chapitre suivant, nous présenterons les contraintes liées à cette méthodologie et la manière d'adapter les raisonnements lorsque, dans la pratique, ces contraintes ne sont pas respectées.

1.1. Forme des spécifications.(1)

1.1.1. Schéma de spécification.

Une spécification se compose d'une **en-tête**, d'une **spécification des paramètres**, d'une **précondition** et d'une **postcondition**.

◇ 1 ◇ Une *en-tête* est de la forme :

$p(Par_1 , \ldots , Par_N)$

où - p est le nom de procédure,

- $Par_1 , \ldots , Par_N$ sont les paramètres

(identificateurs ne figurant de préférence pas dans le texte exécutable de la procédure),

◇ 2 ◇ Une *spécification des paramètres* est de la forme :

- données : $Par_i^1 , \ldots , Par_i^n$,(2)
- résultats : $Par_j^1 , \ldots , Par_j^m$,
- données modifiées : $Par_k^1 , \ldots , Par_k^p$,

+ liste des fichiers modifiés.

---

1. spécifications applicables à certains types de procédures.

2. Par_i^1 doit être lu " Par " indice " $_i^1$ " , et non " Par^1 " indice i.

Chaque paramètre figure dans une et une seule des trois listes.

(n + m + p = N)

Paramètres données : Si Par_i figure dans la liste des paramètres de données, cela signifie que, lors de l'appel de la procédure avec des termes t_1 , ... , t_N , le terme t_i doit être un terme de base et Par_i désigne ce terme dans la spécification de la procédure.

Paramètres résultats : Si Par_i figure dans la liste des paramètres de résultats, le terme t_i doit être une variable et Par_i désigne le $i^{ème}$ résultat de l'appel de procédure.

De plus, tous les termes t_j^1 , ... , t_j^m doivent être des variables distinctes, ne figurant pas dans les termes t_k^1 , ... , t_k^p (ceci pour simplifier les raisonnements relatifs à l'unification des têtes de clauses avec p(t_1 , ... , t_N) : avec les conditions ci-dessus, on peut ignorer les termes t_j^1 , ... , t_j^m dans ces raisonnements).

Paramètres données modifiées : Dans ce cas, il n'y a pas de condition sur le terme t_i et Par_i désigne $\mathbf{t_i}$ dans la précondition de la spécification,

$\mathbf{t'_i}$ dans la postcondition.

(même chose pour les noms de fichiers)

Dans la spécification des paramètres, on peut rajouter des concepts 'virtuels'(3), ou donner des conventions standard(4), et s'en servir dans la précondition et la postcondition.

---

3. comme, par exemple, dans l'application qui sera présentée plus loin, le concept de s.m.r.l. (suite de mots restant à lire),

4. comme, dans la même application, la convention sur les caractères courants $Ccour_i$, $Ccour_f$,...

◇ 3 ◇ La *précondition* énonce les conditions à vérifier par les termes t_1 , ... , t_N (et, éventuellement par les fichiers) pour que l'exécution ait un sens (5)

La précondition ne doit rien affirmer sur les termes t_j^1 , ... , t_j^m (fixés plus haut).

Dans la précondition, les termes t_i^1 , ... , t_i^n et t_k^1 , ... , t_k^p sont désignés par les noms Par_i^1 , ... , Par_i^n et Par_k^1 , ... , Par_k^p.

La précondition sert également à fixer des notations pour les valeurs initiales des fichiers et des données modifiées t_k^1 , ... , t_k^p .

◇ 4 ◇ La *postcondition* définit les résultats t'_1 , ... , t'_N et l'état final des fichiers. Elle ne doit rien affirmer sur les termes $t'_i{}^1$, ... , $t'_i{}^n$ (car $t'_i{}^j = t_i^j$, étant donné que ces termes étaient des données.)

Dans la postcondition, les noms Par_j^1 , ... , Par_j^m et Par_k^1 , ... , Par_k^p désignent les termes $t'_j{}^1$, ... , $t'_j{}^m$ et $t'_k{}^1$, ... , $t'_k{}^p$.

Et si la procédure à spécifier réussit toujours, cela suffira, sinon, on indiquera explicitement les conditions de réussite et d'échec.

1.1.2. Exemple : Spécification de la procédure append

(une procédure peut avoir plusieurs spécifications)

en-tête : append (L_1 , L_2 , _R)

données : L_1 , L_2

résultats : _R

pré : L_1 , L_2 sont des listes.

post : _R = $L_1 \diamond L_2$ (concaténation de L_1 avec L_2).

remarque : bien que concise cette spécification est absolument précise.

---

5. La notion de *précondition* est différente de celle utilisée pour définir les prédicats de D.Log. En effet, dans ce cas, on prenait en compte l'ensemble des conditions pour que l'exécution 'réussisse' et pas seulement 'ait un sens'.

1.2. Forme des raisonnements.

On donne ici un schéma de démonstration applicable en principe pour démontrer qu'une procédure spécifiée comme indiqué plus haut est correcte par rapport à sa spécification.

En pratique ce schéma devra souvent être 'adapté' . Il ne représente pas une démarche immuable, mais un principe général à suivre.

1.2.1. Préliminaires.

Soit une **déclaration de procédure p** : clause_1 ... clause_n

Chaque clause clause_i étant de la forme :

$$p(T_1 , \ldots , T_m) :- S .$$

où - les T_i sont des termes,

- S est une suite de buts.

On suppose que la procédure est **récursive**, mais qu'il n'y a pas de récursivité croisée. En cas de récursivité croisée, il faudrait faire une démonstration simultanée de la correction des diverses procédures concernées. Une procédure non récursive est un cas particulier de procédure récursive.

La démonstration de correction suppose l'existence d'un **programme P** contenant des déclarations de toutes les procédures utilisées par la procédure p et respectant des spécifications (démontrées à part).

On suppose que la procédure p est **spécifiée** comme indiqué ci-dessus.

1.2.2. Schéma de démonstration.

Le **principe** est de démontrer la correction de l'appel de procédure par induction sur les données . Par données, on entend les valeurs prises par les paramètres données (+ éventuellement données modifiées) lors d'un appel de la procédure.

Ceci suppose qu'on ait fixé dans la spécification (en général, dans la précondition) l'ensemble des valeurs possibles de ces paramètres et qu'on ait fixé sur cet ensemble une *relation bien fondée* '<' .

En pratique, on raisonnera le plus souvent :

- soit sur la structure des termes,
- soit sur une fonction entière des données.

Hypothèse simplificatrice :

On supposera ici que **l'appel** de la procédure **réussit toujours** et qu'**une seule clause est choisie pour être exécutée** : la sélection de cette clause se base uniquement sur le *matching* des têtes de clauses et des arguments.

Autrement dit : on exécute la première clause dont la tête *matche*, et celle-ci réussit.

Cette hypothèse revient à considérer la déclaration comme quelque chose de semblable à la forme "COND" en LISP .

Soit n , le nombre de clause de la déclaration.

Le raisonnement comporte n cas dont chacun correspond à l'exécution d'une clause particulière.

A. Schéma du raisonnement.

Cas 1 (exécution de la clause $clause_1$)

.

.

.

Cas n (exécution de la clause $clause_n$)

B. Contenu du raisonnement pour le cas i. ($1 <= i <= n$)

Le $i^{ème}$ cas du raisonnement est traité comme suit :

1. On énonce la **condition C_i** vérifiée par les arguments pour que le cas i soit le cas à considérer.

(On a : précondition $==> C_1 \vee C_2 \vee \ldots \vee C_n$

et $(C_i \& C_j)$ = faux, si $i \neq j$)

2. On montre que si C_i est vrai, l'en-tête des clauses $clause_1$, ..., $clause_{i-1}$ **ne peut être unifiée** avec l'appel de la procédure (ce qui justifie que les corps de ces clauses ne seront pas exécutés).

3. Ensuite, on montre que si C_i est vrai, la tête de la clause clause$_i$ **peut être unifiée** avec l'appel de la procédure.

Note : pour les points 2. et 3., seuls les paramètres données et données modifiées sont à prendre en compte. D'après les hypothèses faites sur les valeurs des paramètres résultats celles-ci sont des variables distinctes ne figurant pas dans les valeurs des données modifiées : elles peuvent donc être unifiées avec n'importe quoi et cette unification n'a aucune influence sur celle des autres arguments.

D'autre part, l'unification d'un terme donnée (argument effectif) avec le terme correspondant de la clause (argument formel) est un simple problème de *pattern matching*. Il n'y a lieu de faire de l'unification au sens général du terme que lorsqu'il y a des paramètres données modifiées.

4. On détermine les **valeurs initiales des variables du contexte** dans lequel le corps S_i de la clause sera exécuté.

5. On **exécute 'symboliquement'** le corps S_i de la clause dans le contexte (détaillé plus loin).

Note : si S_i contient un appel récursif de p , il faut montrer, lors de l'exécution symbolique, que les arguments effectifs de l'appel récursif sont '<' , selon la relation bien fondée choisie, que les arguments effectifs considérés au départ du raisonnement.

6. On calcule les **valeurs finales des paramètres données modifiées et résultats** en instanciant les arguments formels correspondant aux valeurs finales des variables du contexte.

7. On montre que ces valeurs finales **vérifient la postcondition** de la spécification.

C. Exécution symbolique d'une suite de buts dans un contexte.

Si S s'écrit B_1 , ... , B_m , on exécute successivement B_1 dans le contexte initial, B_2 dans le contexte résultant, et ainsi de suite.

Chaque état intermédiaire peut être décrit par une expression de la forme :

" $X_1 = t_1$, ... , $X_n = t_n$ où ...condition... "

dans laquelle : - les X_i sont les variables du contexte,

- les t_i , des expressions décrivant les valeurs de ces variables,
- '...condition...', une série de conditions vérifiées par les t_i .

D. Excécution symbolique d'un but dans un contexte.

L'état initial du contexte est décrit par une expression de la forme :

" $X_1 = t_1$, ... , $X_n = t_n$ où ...condition... "

Le **but à exécuter** est de la forme

$p(T_1 , ... , T_m)$

On **instancie 'symboliquement'** les termes T_1 , ... , T_m avec les valeurs t_1 , ... , t_n des variables X_1 , ... , X_n . Des simplifications peuvent être réalisées en tenant compte de la condition "...condition...".

On obtient ainsi les termes instanciés T'_1 , ... , T'_m vérifiant une condition "**...condition'...** ".

On fait référence à la spécification de la procédure p , pour déterminer 'symboliquement' le résultat de l'appel :

$p(T'_1 , ... , T'_m)$.

Pour cela , il faut **vérifier** :

1. que les T'_i correspondant à des paramètres de données représentent des termes de base,
2. que les T'_i correspondant à des paramètres de résultats représentent des variables ne figurant pas dans les autres termes T'_j ($j \neq i$)
3. que les T'_i correspondant à des paramètres de données et données modifiées vérifient la précondition de la procédure.

Ensuite, la spécification de la procédure permet de déterminer la forme des résultats :

T''_1 , ... , T''_m vérifiant la condition "...condition''..."

L'unification 'symbolique' de ces résultats avec T_1 , ... , Tm conduit à modifier les valeurs de certaines variables du contexte, ce qui conduit à la description du **nouvel état du contexte** par une expression de la forme :

" $X_1 = t'_1$, ... , $X_n = t'_n$ où condition''' ".

Remarque :

L'unification en question se fait de façon simplifiée pour les i correspondant à des paramètres résultats ou données :

- si Par_i est un paramètre donnée : rien à faire ($T''_i = T'_i$).
- si Par_i est un paramètre résultat : T_i est une variable du contexte et il suffit d'écrire : $T_i = T''_i$.

E. Simplifications possibles.

On peut donner des **conventions supplémentaires** pour simplifier l'expression des différents états du contexte correspondant aux étapes de l'exécution d'une suite de buts.

Par exemple :

◇ Dans l'état initial, on peut se contenter de mentionner les valeurs des variables instanciées à une valeur autre qu'elle-même (sous-entendu : une variable non citée a pour valeur elle-même).

◇ Dans un état ultérieur, on peut se contenter de mentionner les valeurs des variables ayant changé de valeur (sous-entendu : une variable non citée n'a pas changé de valeur).

1.2.3. Exemple : Démonstration de la correction de append

Soit la procédure suivante :

```
append ( [] , _X , _X ).
append ( [ _X | _Y ] , _Z , [ _X | _W ] ) :-
        append ( _Y , _Z , _W ).
```

Démontrons qu'elle est correcte par rapport à la spécification donnée au paragraphe 1.1.2.

Démonstration par induction sur la structure de L_1 .

Cas 1. $L_1 = []$

a. L'appel de la procédure *matchera* avec la tête de la première clause car $L_1 = []$ et $\_X$ *matche* avec n'importe quoi.

b. Comme le corps de cette clause est la suite vide, l'état initial et l'état final du contexte coïncident et sont tels que $\_X = L_2$.

c. Le résultat sera donc $\_R = L_2 = []\diamond L_2 = L_1\diamond L_2$.

Cas 2. **L_1 est de la forme [t | L'_1]**

où t est un terme de base et L'_1 est une liste.

a. L'appel de la procédure ne *matchera* pas avec la tête de la première clause car $L_1 \neq []$.

Il *matchera* avec la tête de la seconde car [$\_X$ | $\_Y$] *matche* avec [t | L'_1] , donnant $\_X = t$ et $\_Y = L'_1$ et $\_Z$ matche avec L_2 , donnant $\_Z = L_2$.

b. L'exécution du but **append**($\_Y$, $\_Z$, $\_W$) dans ce contexte provoque l'appel de procédure :

append (L'_1 , L_2 , $\_W$)

qui est compatible avec la spécification puisque L'_1 et L_2 sont des termes de base et $\_W$ est une variable.

Le résultat de l'appel récursif de la procédure sera donc, par hypothèse d'induction : $R = L'_1\diamond L_2$.

et le contexte final tel que : $\_W = L'_1\diamond L_2$.

c. Finalement, le résultat de la procédure sera :

$$R = [\ t \mid L'_1 \diamond L_2\ ] = [\ t \mid L'_1\ ] \diamond L_2 = L_1 \diamond L_2 .$$

c.q.f.d.

1.3. Forme des documentations.

1.3.1. Schéma d'une procédure documentée.

◇ Si une procédure est de la forme :

clause$_1$
.
.
.
clause$_n$

la **procédure documentée** correspondante sera de la forme :

clause-doc$_1$
.
.
.
clause-doc$_n$

Et, pour les procédures qui peuvent échouer, on ajoute, après clause-doc$_n$, une **condition d'échec** qui spécifie la condition sous laquelle il y a échec :

{ condition d'échec : CE }

◇ Et si une clause clause$_i$ est de la forme :

p(T_1 , ... , T_n) :- B_1 , ... , B_m .

la **clause documentée** clause-doc$_i$ correspondante sera de la forme :

(C) *p(T_1 , ... , T_n)* :- (CD)
B_1 (CI) .
. .
. .
B_m (CF) .

(P)

où C = condition de sélection de la clause,

CD = description contexte initial,

CI = description état intermédiaire du contexte [6],

CF = description état final du contexte [7],

P = description des valeurs finales des paramètres résultats et données modifiées (ou, à défaut, description de *ce qu'on sait qui est vrai* lorsque l'exécution de cette clause est terminée).

Et, pour les clauses qui peuvent échouer, la condition de sélection est suivie d'une autre condition (**condition de réussite**) :

(réussi si : C')

et **chaque but pouvant échouer** de la clause est précédé d'une condition de réussite : (réussi si : C_i)

1.3.2. Exemple : Documentation de la procédure append

Documentons cette procédure par rapport à la spécification donnée au paragraphe 1.1.2.

(L_1 = [])

append ([] , _X , _X) (_X = L_2) .

(_R = L_2 = []◇L_2 = L_1◇L_2)

(L_1 = [t | L'_1] où t est un terme de base et L'_1 est une liste)

append ([_X / _Y] , _Z , [_X / _W]) :-

(_X = t , _Y = L'_1 , _Z = L_2)

append (_Y , _Z , _W) (_W = L'_1◇L_2) .

(_R = [t | L'_1◇L_2] = [t | L'_1]◇L_2 = L_1◇L_2)

---

6. On pourra se contenter de ne donner que les valeurs des variables de contexte qui ont été modifiées depuis la description précédente.

7. idem.

2. APPLICATION DE LA METHODOLOGIE.

2.1. Présentation du programme.

Nous allons appliquer la méthodologie proposée à un programme qui a été construit au cours du stage préliminaire de ce mémoire.

Ce programme consiste en une compilation entre un langage de spécifications et un formalisme de représentation.
Une brève description en est donnée ci-dessous, mais on pourra consulter en annexe un rapport contenant la présentation détaillée du problème, les spécifications et le texte complet du programme .

2.1.1. Le problème.

Ce programme s'inscrit dans le développement d'un langage de modélisation de concepts, projet du département de recherche de la société B.I.M.

L'objet du programme est de réaliser la compilation d'un texte écrit en CML[1] en un formalisme permettant de gérer plus facilement les informations contenues dans ce texte, par une utilisation de la logique de Prolog. Ce formalisme s'appelle Pif[2], et est constitué de faits Prolog intégrables dans la base de faits d'un autre programme.

Le programme de compilation se décompose comme suit :

- analyse lexicale :

  A partir d'un fichier de caractères contenant un texte CML, décomposer ce texte en phrases correspondant aux différents objets CML.

  Par phrase, repérer la présence éventuelle de caractères étrangers à CML, fournir un message pour chacun de ces caractères, et donner le type de chaque mot de la phrase.

---

1. a Conceptual Modelling Language for software engineering.

2. Prolog Internal Formalism.

- analyse syntaxique :

  Pour chaque phrase, repérer la présence d'erreurs éventuelles par rapport à la syntaxe du langage CML, et fournir une liste de ces erreurs.

- analyse sémantique :

  Pour chaque objet syntaxiquement correct, vérifier la correction sémantique et fournir une listes des erreurs éventuelles.

- décomposition en faits :

  Dans le cas où l'analyse complète du contenu du fichier de départ n'a révélé aucune erreur syntaxique ou sémantique pour aucun objet du modèle, fournir une liste de faits Prolog, dans le formalisme Pif, correspondant au modèle de départ.

2.1.2. Notions de base.

Afin de simplifier le travail de spécification, et pour obtenir des spécifications claires, concises et non ambiguës, un certain nombre de notions sont définies au préalable.

Le contenu du fichier d'entrée du programme pouvant être considéré selon plusieurs niveaux d'abstraction (fichier = suite de phrases, phrase = suite de mots, mot = suite de caractères), les notions définies ont été classées selon ces niveaux :

A. Au niveau *caractères*

1. **Caractère normal.**

   Un caractère normal est n'importe quel caractère accepté par la syntaxe de CML, c'est-à-dire
   - les lettres majuscules et minuscules
   - les chiffres,
   - '&', '/', '?', '@', '"', '[', ']', '_', '-', '\'.

2. **Caractère skipable = caractère d'espacement.**

   Un caractère skipable est un caractère dont on peut ne pas tenir compte à l'analyse lexicale, c'est-à-dire 'space' , 'tab' , 'eoln' (caractère de saut de ligne).

3. **Caractère spécial.**

Un caractère spécial est un caractère qui, dans le contexte de l'analyse lexicale, nécessite un traitement spécial,

c'est-à-dire '(' , ':' , ',' , ')' et le caractère qui a été choisi de manière interne à ce programme pour signaler la fin de fichier (0 ASCII).

4. **Caractère délimiteur.**

Un caractère délimiteur est un caractère qui permet de décomposer un texte en entités (mots) c'est-à-dire un caractère d'espacement ou un caractère spécial.

5. **Caractère illégal.**

Un caractère illégal est un caractère non normal et non délimiteur.

Les caractères de ce type ne seront pris en compte dans aucun mot et provoqueront un message d'erreur.

Remarque :

Le type d'un caractère dépend également du contexte (défini ci-dessous) dans lequel il se trouve. Par exemple, le caractère ')' est un caractère spécial s'il marque la fin d'un commentaire, et est un caractère illégal sinon.

B. Au niveau *mots*

Soit α, une suite de caractères constituant les *données restant à lire par le programme*.

1. **Contexte d'un caractère c dans une chaîne α** .

Une occurrence d'un caractère dans α appartient à un et un seul des trois contextes suivants.:

a. c est dans le contexte *'termes'* pour une chaîne α ,

ssi c figure dans un terme (au sens Prolog du mot) précédé du caractère ':' (non compris dans ce contexte) et suivi du premier ' . ‹eoln› ' (compris dans ce contexte) dans la chaîne α ,

b. c est dans le contexte *'non significatif'* pour une chaîne α ,

ssi - soit c figure entre un caractère '(' et le premier caractère ')' qui le suit dans α ('(' et ')' compris dans ce contexte),

- soit c est un caractère illégal (c'est-à-dire non normal et non délimiteur),
- soit c est un caractère skipable,

c. c est dans le contexte *'significatif'* pour une chaîne α ,

ssi c n'est pas dans les deux autres contextes pour la chaîne α .

2. **Caractère significatifs**.

On appelle caractère significatif, un caractère appartenant au contexte *'significatif'* (3)

3. **Mots**.

Un mot est, dans le contenu de α , un groupe de caractères significatifs (sauf ',' et ':') directement contigus ou éventuellement séparés deux-à-deux par un ou plusieurs caractères illégaux, mais uniquement par eux.
Le caractère ',' forme également un mot à lui seul, ainsi qu'un terme appartenant au contexte 'termes', précédé de ':' et suivi de ' . ‹eoln› '.

Remarque :

Les caractères illégaux sont retirés des mots lors de l'analyse lexicale(4).

4. **Caractère courant**.

Le programme à construire ne manipule jamais qu'un fichier d'entrée qui est le texte source à compiler. Ce fichier a le nom logique 'in'.

---

3. autrement dit, un caractère normal ou ',' ou ':' .

4. d'où la définition de *mot* de la page 14 du rapport en annexe.

Le dernier caractère lu dans ce fichier sera traité par le programme non comme un caractère, mais comme un entier égal au code ASCII de ce caractère. Et c'est cette valeur qu'on appellera caractère courant.

Cependant, comme il n'y a pas de notion de variable globale en Prolog, il faudra préciser à chaque instant 'où se trouve cette valeur' :

- lors d'un appel de procédure, ce devra être la valeur d'une des données de la procédure,
- à la fin de cet appel, ce devra être la valeur d'un résultat de la procédure.

Donc, dans l'analyse lexicale, le caractère courant sera toujours le prochain caractère qui sera traité, et il faudra préciser dans les raisonnements où il se trouve.

Remarque :

L'analyse lexicale sera construite de telle façon qu'à tout moment de son exécution, le caractère courant pourra prendre :

- soit la valeur d'un caractère significatif,
- soit la valeur nulle, s'il n'y a plus de caractère à lire dans le fichier.

5. Chaîne courante.

Si α' désigne la suite des données restant à lire par le programme, et si c désigne le caractère courant, la chaîne $c\alpha'$, notée α , est appelée chaîne courante (ou chaîne restant à traiter).

Et si c = o , on pose α est la chaîne vide.

6. Suite des mots restant à lire.

La suite des mots restant à lire (s.m.r.l.) est égale à m_1 , ... , m_n ($n >= 1$)

ssi la chaîne courante est de la forme $m_1\lambda_2 \ldots m_n\lambda_{n+1}$,

où - les λ_i sont des suites de caractères non significatifs, suites pouvant être vides, sauf si elles séparent deux mots qui ne sont ni la virgule, ni un terme.

- les m_i sont des mots.

Si la chaîne courante est vide, la s.m.r.l. est vide.

C. Au niveau *phrases*

1. **Mot typé.**

Un mot typé est un terme de la forme ' type(mot) ' , avec comme argument un mot et, comme tête, le type de ce mot, qui dépend du mot lui même :

- si le mot est une suite de caractères normaux constituant un mot clé, le type de ce mot est *key*.
- si le mot est une suite de caractères normaux constituant une catégorie, le type de ce mot est *cat*.
- si le mot est une autre suite de caractères normaux, le type de ce mot est *mot*.
- si le mot est le caractère spécial ',' , le type de ce mot est *del*.
- si le mot est un terme Prolog [5], le type de ce mot est *val*.

2. **Phrase.**

Une phrase est une suite de mots qui, si elle est conforme à la syntaxe de CML, constitue un *objet CML*.

3. **Liste de mots typés.**

L'interface entre l'analyse lexicale et l'analyse syntaxique est réalisé au moyen de listes (au sens Prolog) de mots typés.

2.2. Justification du programme.

Il est possible de classer les procédures en plusieurs catégories en se basant sur certains critères dont les principaux sont :

---

5. On a retiré ici les ' : ' et ' . ‹eoln› ' qui accompagnaient le terme.

- la possibilité d'échec (procédures qui réussissent toujours ou procédures qui peuvent échouer),
- le choix de la clause à exécuter (présence ou non de buts qui peuvent échouer dans une clause, ce qui fait échouer la clause et remet en cause le *matching*),
- récursivité (procédures récursives ou non).
- paramètres (procédures dont la spécification contient des données modifiées ou uniquement des données et résultats).

Il est également possible de multiplier le nombre de ces catégories en combinant plusieurs critères.

Pour illustrer la méthodologie présentée précédemment, nous devions choisir quelques procédures à démontrer et à documenter.

Nous avons effectué ce choix dans le programme de compilation de telle manière à obtenir :

- un échantillon représentatif des différentes catégories ou sous-catégories de procédures,
- des procédures de toutes les parties représentatives du programme.

Mais, afin de permettre de démontrer ces procédures, nous donnons également la spécification des procédures qu'elles utilisent.

Et pour toutes ces procédures, nous préciserons à chaque fois la partie du programme de laquelle elles sont tirées, le numéro de la page où on peut les trouver dans le rapport de stage et, quand cela sera utile, la catégorie à laquelle elles appartiennent.

2.2.1. Procédures de l'analyse lexicale.

A. procédure normcar (page 42 du rapport de stage).

(procédure qui peut échouer et dont certaines clauses peuvent échouer)

1. spécification.

en-tête : normcar(i)

donnée : i

pré : i est un entier.

post : si i est le code ASCII d'un caractère normal,

alors la procédure se termine avec échec = 'faux'

(autrement dit : l'appel réussit [6]).

sinon elle se termine avec échec = 'vrai'.

(autrement dit : l'appel échoue).

remarque :

Le but de cette procédure est de cerner la notion de caractère normal.

Donc, conformément à sa définition, dans la spécification, dire que i est le code ASCII d'un caractère normal est équivalent à dire que :

```
   65 <= i<= 91   ( code ASCII d'une lettre majuscule ),
ou 97 <= i<= 122  ( code ASCII d'une lettre minuscule ),
ou 48 <= i<= 57   ( code ASCII d'un chiffre ),
ou i ∈ ( 38, 39, 45, 47, 63, 64, 91, 92, 93, 95 )
        ( code ASCII de & , ' , - , / , ? , @ , [ , \ , ] , _ ),
```

programme :

```
normcar ( _X ) :- _X >= 65 ,
                  _X <= 91 .

normcar ( _X ) :- _X >= 97 ,
                  _X <= 122 .

normcar ( _X ) :- _X >= 48 ,
                  _X <= 57 .

normcar ( 38 ) .
normcar ( 39 ) .
normcar ( 45 ) .
normcar ( 47 ) .
normcar ( 63 ) .
normcar ( 64 ) .
normcar ( 91 ) .
normcar ( 92 ) .
normcar ( 93 ) .
normcar ( 95 ) .
```

2. documentation.

---

6. Et dorénavant, nous emploierons plutôt cette formulation :
l'appel réussit ou échoue.

{ } { réussi si : 65 <= i <= 91 }

normcar (_X) :- { _X = i }

{ réussi si : 65 <= i } *_X >= 65* { 65 <= i } ,

{ réussi si : i <= 91 } *_X <= 91* { i <= 91 } .

{ 65 <= i <= 91 }

{ i < 65 ou i > 91 } { réussi si : 97 <= i <= 122 }

normcar (_X) :- { _X = i }

{ réussi si : 97 <= i } *_X >= 97* { 97 <= i } ,

{ réussi si : i <= 122 } *_X <= 122* { i <= 122 } .

{ 97 <= i <= 122 }

. . . .

{ i = 38 }

normcar (38) { } .

.
.
.

{ i = 95 }

normcar (95) { } .

{ condition d'échec : non(65 <= i <= 91 ou 97 <= i <= 122 ou
48 <= i <= 57 ou i ∈ { 38,39,45,...,95 }) }

B. procédure give_car (page 62 du rapport de stage).

en-tête : give_car(_C)

résultat : _C

donnée modifiée : 'in'

pré : 'in', ouvert en lecture, de contenu accessible α .

post : si $\alpha = c\alpha'$ alors _C est égal au code ASCII de c ,

et le contenu accessible du fichier est égal à α' ,

et si α est vide, alors _C = 0.

C. procédure give_term (page 64 du rapport de stage).

en-tête : give_term(_T)

résultat : _T

donnée modifiée : 'in'

pré : 'in', ouvert en lecture, de contenu accessible α .

post : si $\alpha = td\alpha'$ où t est un terme, et d est la séquence ' . ‹eoln› ' ,

alors ◇ _T = t ,

◇ le contenu accessible de 'in' = α' ,

sinon ◇ _T = 'wrong term' .

◇ le contenu accessible de 'in' est un suffixe de α ,

remarque :

Cette spécification est incomplète car on n'a pas précisé exactement *ce qui se passe* en cas d'erreur, c'est-à-dire lorsque le contenu du fichier de données 'in' contient des fautes ou des termes incorrects.

Cela n'est pas gênant car, dans les démonstrations qui concernent l'analyse lexicale, nous nous concentrerons sur les cas où il n'y a pas d'erreur.

D. procédure find_good_car (page 63 du rapport de stage).

en-tête : find_good_car(C , _GC)

donnée : C

donnée modifiée : 'in'

résultat : chct : chaîne courante
(avec le caractère courant dans _GC)

pré : ◇ C = code ASCII d'un caractère, ou 0 ,

◇ 'in', ouvert en lecture, de contenu accessible α .

post : si ◇ C est le code ASCII du caractère c,

◇ λ est le plus grand préfixe de la chaîne $c\alpha$, notée α' , dont tous les caractères appartiennent au contexte 'non significatif' ,

◇ $\alpha' = \lambda\alpha''$,

alors chct = α'' ,

et si C = 0, alors _GC = 0 .

E. procédure **typage** (page 61 du rapport de stage).

en-tête : typage (M , _MT)

donnée : M

résultat : _MT

pré : M est un mot sauf une virgule ou un terme.

post : _MT est le mot typé d'argument M .

F. procédure **traiter_reste** (page 61 du rapport de stage).

en-tête : traiter_reste (C_i , _M' , $\_Ccour_f$)

donnée : C_i

donnée modifiée : 'in'

résultats : _M' , chct : chaîne courante
(avec le caractère courant dans $\_Ccour_f$)

pré : ◇ C_i = code ASCII d'un caractère c , ou 0 ,

◇ 'in', ouvert en lecture, de contenu accessible α .

post : si ◇ C_i est le code ASCII du caractère c,

◇ la chaîne cα , notée α' , est égale à mλα'' où m est une suite éventuellement vide de caractères normaux[7] et λ est la plus longue suite de caractères non significatifs (suite non vide si α'' commence par un caractère normal),

alors ◇ _M' = la liste des codes ASCII des caractères formant m ,

◇ chct = α'' ,

et si C_i = 0 , alors $\_Ccour_f$ = 0 et _M' est la liste vide.

---

7. On ne tiendra pas compte dans ce cas de la possibilité d'avoir des caractères illégaux dans un mot.

G. procédure traiter_phrase (page 59 du rapport de stage).

(procédure récursive et qui réussit toujours)

1. spécification.

en-tête : traiter_phrase ($Ccour_i$, Mcour1 , _Phrase , $_Ccour_f$)

données : Mcour1 : mot typé,

$Ccour_i$: caractère courant,

résultats : _Phrase , $_Ccour_f$: caractère courant,

donnée modifiée : smr1 : suite de mots restant à lire,

pré : Mcour1 = $\mu(m_1)$, smr1 = m_2 , ... , m_n

où m_1 ... m_n sont des mots (n >= 1).

post : _Phrase = [$\mu(m_1)$, ... , $\mu(m_i)$] , smr1 = m_{i+1} , ... , m_n

où i est le plus petit entier tel que (1 <= i <= n)

et (mi = END ou i = n) .

programme :

```
traiter_phrase( 0 , _mot_type, [ _mot_type ] , 0) .

traiter_phrase( _car_contr, key(END), [ key(END) ] , _car_contr) .

traiter_phrase( _car_contr, _mot_type, [ _mot_type | _reste ] , _car_next) :-
        lire_mot( _car_contr , _n_mot_type , _car_suiv ) ,
        traiter_phrase( _car_suiv , _n_mot_type , _reste , _car_next ) .
```

2. démonstration par induction sur n.

Les cas suivants sont possibles :

Cas 1. **n = 1**

a. Alors, $Ccour_i$ = 0 car smr1 est vide, ce qui équivaut à dire que le caractère courant vaut 0 . Donc l'appel de la procédure *matche* avec la tête de la première clause car 0 *matche* avec 0 et _mot_type *matche* avec $\mu(m_1)$.

b. Comme le corps de cette clause est la suite vide, l'état initial et l'état final du contexte coïncident et sont tels que _mot_type = $\mu(m_1)$.

c. L'exécution de la procédure se termine avec

_Phrase = [μ (m_1)] et _Ccour$_f$ = 0

(et smrl reste égale à λ).

Cas 2. **n > 1 et i = 1**

a. Alors, m_1 = END et Ccour$_i$ ≠ 0, donc l'appel de la procédure ne *matche* pas avec la tête de la première clause.

Il *matche* avec la tête de la seconde car _car_contr *matche* avec n'importe quoi, et $\mu(m_1)$ = key(END) .

b. Comme le corps de cette clause est la suite vide, l'état initial et l'état final du contexte coïncident et sont tels que _car_contr = Ccour$_i$.

c. L'exécution de la procédure se termine avec

_Phrase = [key(END)] et smrl = m_2 , ... , m_n

Cas 3. **n > 1 et i > 1**

a. Alors, Ccour$_i$ ≠ 0 et m1 ≠ END , donc l'appel de la procédure ne *matche* pas avec la tête de la première et de la deuxième clause.

Il *matche* avec la tête de la troisième car _car_contr et _mot_type *matchent* avec n'importe quoi.

b. Le contexte initial est : _mot_type = $\mu(m_1)$ et smrl = m_2 , ... , m_n

(_car_contr = Ccour$_i$).

- L'exécution du but **lire_mot**(_car_contr , _n_mot_type , _car_suiv) dans ce contexte provoque l'appel de procédure :

lire_mot(Ccour$_i$, _n_mot_type , _car_suiv) ,

qui est compatible avec sa spécification puisque Ccour$_i$ est un terme de base, les deux autres termes sont des variables et smrl n'est pas vide.

L'exécution se termine avec _n_mot_type = $\mu(m_2)$

et smrl = m_3 , ... , m_n (caractère courant : _car_suiv).

- L'exécution, dans ce contexte, du but

traiter_phrase(_car_suiv , _n_mot_type , _reste , _car_next)

provoque l'appel de procédure :

traiter_phrase(c , $\mu(m_2)$, _reste , _car_next) .

où c = _car_suiv

qui est compatible avec la spécification puisque c et $\mu(m_2)$ sont des termes de base.

Par hypothèse d'induction, l'appel récursif de la procédure se terminera avec _reste = [$\mu(m_2)$, ... , $\mu(m_i)$] , smrl = m_{i+1} , ... , m_n (caractère courant : _car_next)

où i est le plus petit entier tel que (1 <= i <= n)

et (mi = END ou i = n).

c. L'exécution de la procédure se termine avec

_Phrase = [μ (m_1) | [μ (m_2) , ... , μ (m_i)]]

= [μ (m_1) , μ (m_2) , ... , μ (m_i)]

et smrl = m_{i+1} , ... , m_n

(et $\_Ccour_f$ est le caractère courant).

c.q.f.d.

3. documentation.

(n = 1)

traiter_phrase(0 , _mot_type, [_mot_type] , 0)

(_mot_type = $\mu(m_1)$) .

(_Phrase = [μ (m_1)] et smrl = λ)

(n > 1 et i = 1)

traiter_phrase(_car_contr, key(END), [key(END)] , _car_contr)

(_car_contr = $Ccour_i$) .

(_Phrase = [key(END)] = [μ (m_1)] et smrl = m_2 , ... , m_n)

(n > 1 et i > 1)

traiter_phrase(_car_contr, _mot_type, [_mot_type / _reste] , _car_next) :-

(_mot_type = $\mu(m_1)$ et _car_contr = $Ccour_i$)

lire_mot(_car_contr , _n_mot_type , _car_suiv)

(_n_mot_type = $\mu(m_2)$ et

smrl = m_3 , ... , m_n (car. cour. : _car_suiv)) ,

traiter_phrase(_car_suiv , _n_mot_type , _reste , _car_next)

(_reste = [$\mu(m_2)$, ... , $\mu(m_i)$] et

smrl = m_{i+1} , ... , m_n (car. cour. : _car_next)) .

(_Phrase = [$\mu(m_1)$, ..., $\mu(m_i)$] et smrl = m_{i+1} , ... , m_n

(car. cour. : $\_Ccour_f$))

H. procédure **lire_mot** (page 60 du rapport de stage).

(procédure non récursive, et qui réussit toujours)

1. spécification.

en-tête : lire_mot ($Ccour_i$, _Mcour , $\_Ccour_f$)

donnée : $Ccour_i$: caractère courant,

donnée modifiée : chct : chaîne courante,

résultats : $\_Ccour_f$: caractère courant,

_Mcour : mot typé,

pré : chct = $m\lambda\alpha'$ où - m est un mot,

- λ est une suite de caractères non significatifs, non vide, si m n'est ni ',' ni un terme et si α' ne commence ni par ',' ni par ':' ,
- α' commence par un caractère significatif ou est vide.

post : ◇ _Mcour = $\mu(m)$,

◇ chct = α' .

programme :

lire_mot(44 , del(',') , _car_suiv) :-

```
                give_car( _car ) ,
                find_good_car( _car , _car_suiv ) .

lire_mot( 58 , val(_value) , _car_suiv ) :-
                give_term( _value ) ,
                give_car( _car ) ,
                find_good_car( _car , _car_suiv ) .

lire_mot( _car_cour , _mot_type , _car_suiv ) :-
                normcar( _car_cour ) ,
                give_car( _car_int ) ,
                traiter_reste( _car_int , _reste_mot , _car_suiv ) ,
                name( _mot , [ _car_cour | _reste_mot ] ) ,
                typage( _mot , _mot_type ) .
```

2. démonstration.

Les cas suivants sont possibles :

m est soit ',' ,
soit un terme,
soit un autre mot (une suite de caractères normaux).

Si m = ',' ,alors $Ccour_i$ = 44 : code ASCCI du caractère ',' (cas 1),

si m = un terme ,alors $Ccour_i$ = 58 : code ASCCI du caractère ':' (cas 2),

si m = un autre mot , alors $Ccour_i$ = code ASCCI d'un caractère normal (cas 3).

Cas 1. m = ','

a. Alors, $Ccour_i$ = 44 et l'appel de la procédure *matche* avec la tête de la première clause car 44 *matche* avec 44 .

b. Le contexte initial est : chct = mλα'

- L'exécution du but **give_car**(_car) dans ce contexte provoque l'appel de procédure :

```
give_car ( _car )
```

qui est compatible avec sa spécification puisque _car est une variable.

L'exécution se termine avec _car = c' où c' est le code ASCII du premier caractère de $\lambda\alpha'$ (ou 0 si $\lambda\alpha'$ est vide), et le contenu accessible de 'in' égal à $\lambda\alpha'$ amputé de son premier caractère c' (si $\lambda\alpha'$ est non vide).

- L'exécution du but **find_good_car**(_car , _car_suiv) dans ce contexte provoque l'appel de procédure :

find_good_car (c' , _car_suiv)

qui est compatible avec sa spécification puisque c' est un terme de base et _car_suiv est une variable.

L'exécution se termine avec chct = α' (car. cour. : _car_suiv).

c. L'exécution de la procédure se termine en renvoyant

_Mcour = del(',') , chct = α '

(et $\_Ccour_f$ est le caractère courant),

ce qui vérifie la postcondition puisque _Mcour = $\mu(m)$ où m = ',' et μ est le type de ce mot, et puisque chct = α'.

Cas 2. **m = un terme**

a. Alors, $Ccour_i$ = 58 , donc l'appel de la procédure ne *matche* pas avec la tête de la première clause.

Il *matche* avec la tête de la seconde car 58 *matche* avec 58 .

b. Le contexte initial est : chct = $m\lambda\alpha'$ où m est de la forme ': t . ‹eoln›'

- L'exécution du but **give_term**(_value) dans ce contexte provoque l'appel de procédure :

give_term (_value)

qui est compatible avec sa spécification puisque _value est une variable.

L'exécution se termine avec _value = t et la suite de données restant à traiter = $\lambda\alpha'$.

- L'exécution des buts **give_car**(_car) et

find_good_car(_car , _car_suiv)

dans ce contexte est similaire au cas 1 et se termine avec chct = α' (caractère courant : _car_suiv).

c. L'exécution de la procédure se termine en renvoyant

_Mcour = val(t) , chct = α '

(et $\_Ccour_f$ est égal au caractère courant),

ce qui vérifie la postcondition puisque _Mcour = μ(m') où m' = t et μ est le type de ce mot, et puisque chct = α' .

Cas 3. **m = un autre mot**

a. Alors, $Ccour_i$ = c , code ASCII d'un caractère normal,

donc l'appel de la procédure ne *matche* pas avec la tête de la première et de la deuxième clause.

Il *matche* avec la tête de la troisième car _car_cour *matche* avec n'importe quoi, donc avec $Ccour_i$.

b. Le contexte initial est : chct = mλα' (caractère courant dans _car_cour)

- L'exécution du but **normcar**(_car_cour) dans ce contexte provoque l'appel de procédure :

normcar (c)

qui est compatible avec sa spécification puisque c est un terme de base. L'exécution ne peut que réussir[8], étant donné la spécification de $Ccour_i$, et se termine sans modification du contexte.

- L'exécution du but **give_car**(_car_int) dans ce contexte provoque l'appel de procédure :

give_car (_car_int)

qui est compatible avec sa spécification puisque _car_int est une variable. L'exécution se termine avec _car_int = c' où c' est le code ASCII du premier caractère qui suit c dans chct (ou 0 si chct = c).

- L'exécution, dans ce contexte, du but

traiter_reste(_car_int , _reste_mot , _car_suiv)

---

8. On aurait donc éventuellement pu supprimer l'appel de normcar dans cette procédure, la présence de cet appel ne se justifiant que par le fait qu'il fait échouer la procédure si elle est appelée en dehors des conditions prévues par sa spécification.

provoque l'appel de procédure :

traiter_reste (c' , _reste_mot , _car_suiv)

qui est compatible avec sa spécification puisque c' est un terme de base et _reste_mot et _car_suiv sont des variables.

L'exécution se termine avec _reste_mot = la liste des codes ASCII des caractères formant m' si m se code en cm' (c' est le premier élément de m'), et avec chct = α' (caractère courant : _car_suiv)

- L'exécution du but **name**(_mot , [_car_cour | _reste_mot]) dans ce contexte provoque l'appel du prédicat primitif :

name (_mot , [c | m'])

qui est compatible avec sa spécification puisque _mot est une variable et [c | m'] est un terme de base.

L'exécution se termine avec _mot = m .

- L'exécution du but **typage**(_mot , _mot_type) dans ce contexte provoque l'appel de procédure :

typage (m , _mot_type)

qui est compatible avec sa spécification puisque m est un terme de base et _mot_type est une variable.

L'exécution se termine avec _mot_type = μ(m) où μ est le type de m .

c. L'exécution de la procédure se termine en renvoyant

_Mcour = μ (m) , chct = α '

(et $\_Ccour_f$ est égal au caractère courant),

ce qui vérifie la postcondition puisque _Mcour = μ(m) où m = une suite de caractères normaux, et μ est le type de ce mot, et puisque chct = α' .

c.q.f.d.

3. documentation.

(**m** = ',')

lire_mot(44 , del(',') , _car_suiv) :-

(chct = mλα' (car. cou. : 44 (ASCII)))

give_car(_car)

(_car = c' où c' est le code ASCII du premier caractère de λα' (ou 0 si λα' est vide), et le contenu accessible de 'in' égal à λα' amputé de son premier caractère c' (si λα' est non vide)) ,

find_good_car(_car , _car_suiv)

(chct = α' (caractère courant : _car_suiv)) .

(_Mcour = del(',') , chct = α ')

(m = un terme)

lire_mot(58 , val(_value) , _car_suiv) :-

(chct = mλα' où m est de la forme ': t . ‹eoln›')

give_term(_value)

(_value = t et la suite de données restant à traiter = λα') ,

give_car(_car)

(_car = c' où c' est le code ASCII du premier caractère de λα' (ou 0 si λα' est vide), et le contenu accessible de 'in' égal à λα' amputé de son premier caractère c' (si λα' est non vide)) ,

find_good_car(_car , _car_suiv)

(chct = α' (caractère courant : _car_suiv)) .

(_Mcour = val(t) , chct = α ')

(m = un autre mot)

lire_mot(_car_cour , _mot_type , _car_suiv) :-

(chct = mλα' (caractère courant dans _car_cour))

normcar(_car_cour) ( ) ,

give_car(_car_int)

(_car_int = c' où c' est le code ASCII du premier caractère qui suit c dans chct (ou 0 si chct = c)) ,

traiter_reste(_car_int , _reste_mot , _car_suiv)

(_reste_mot = la liste des codes ASCII des caractères formant m'
si m se code en cm' (c' est le premier élément de m')) ,

name(_mot , [_car_cour / _reste_mot]) (_mot = m) ,

typage(_mot , _mot_type)

(_mot_type = μ(m) où μ est le type de m) .

(_Mcour = μ (m) , chct = α ')

2.2.2. Procédures de l'analyse syntaxique.

A. procédure find_synchro (page 75 du rapport de stage).

en-tête : find_synchro (Type , L_1 , $\_L_2$)

données : Type , L_1 = liste de mots typés,

résultats : $\_L_2$ = liste de mots typés .

pré : ◇ Type = 'key' , 'cat' ou 'val' ,

◇ $L_1 = [\mu_1(m_1) , \mu_2(m_2) , \ldots , \mu_n(m_n)]$ où $n >= 0$

post : ◇ $\_L2 = [\mu_i(m_i) , \ldots , \mu_n(m_n)]$ où i est le plus petit entier tel que $1 <= i <= n$ et (μ_i = 'key' (si Type = 'key')

ou $\mu_i \in$ ('key' , 'cat') (si Type = 'cat')

ou $\mu_i \in$ ('key' , 'cat' , 'val') (si Type = 'val')),

◇ _L2 = [ ] si un tel i n'existe pas.

B. procédure synt_erreur_i (page 44 du rapport de stage).

en-tête : synt_erreur_i (t_1 , ... , t_{ni}) où i est un entier, et $n_i >= 0$.

(à chaque valeur de i correspond une procédure)

données : t_1 , ... , t_{ni}

donnée modifiée : 'lis' (9)

---

9. Le fichier de nom logique 'lis' est un fichier de sortie du programme qui contient les messages d'erreur. Il est ouvert en écriture pendant toute la durée de l'exécution du programme.

pré : le contenu de 'lis' est β .

post : 'lis' = βM où M est une suite de caractères contenant t_1 , ... , t_{ni} et constituant un *'message d'erreur'*.

C. procédure anal_def (page 70 du rapport de stage).

(procédure qui peut échouer et dont certaines clauses peuvent échouer)

1. spécification.

en-tête : anal_def (Lmt_1 , _mt , _Lmt_2 , _bool)

donnée : Lmt_1 = liste de mots typés

donnée modifiée : 'lis'

résultats : _mt : mot (ou indéfini),

_Lmt_2 : liste de mots typés,

_bool = 'oui' ou 'non' .

pré : ◇ $Lmt_1 = [\mu_1(m_1) , ... , \mu_n(m_n)]$ où $n >= 0$,

◇ le contenu de 'lis' est β .

post : si $Lmt_1 = [key(DEF) , mot(m_2) , ... , \mu_n(m_n)]$ avec $n >= 2$,

alors - _bool = 'oui' ,

- _mt = m_2 ,

- _$Lmt_2 = [\mu_3(m_3) , ... , \mu_n(m_n)]$,

- 'lis' = β (non modifié),

sinon si $Lmt_1 = [key(DEF) , \mu_2(m_2) , ... , \mu_n(m_n)]$ avec $\mu_2 \neq mot$,

et $n >= 2$,

ou $Lmt_1 = [mot(m_1) , \mu_2(m_2) , ... , \mu_n(m_n)]$ avec $n >= 1$,

alors - _bool = 'non' ,

- _$Lmt_2 = [\mu_i(m_i) , ... , \mu_n(m_n)]$ où $2 <= i <= n$, et i est le plus petit entier tel que μ_i = 'key', s'il existe, (sinon _Lmt_2 = []),

- 'lis' = βM ,

sinon si $Lmt_1 = [key(m_1) , \mu_2(m_2) , ... , \mu_n(m_n)]$,

avec n >= 1 (n > 1 si m_1 = DEF),

<u>alors</u> - _bool = 'non' ,

- $\_Lmt_2 = Lmt_1$,

- 'lis' = βM ,

<u>sinon</u> 'lis' = βM , et l'appel échoue .

<u>remarque</u> :

Cette procédure s'inscrit dans un ensemble de procédures qui sont toutes construites de la même façon. Chacune d'entre elles est chargée d'analyser une partie spécifique de la liste de mots typés qu'elle reçoit :

- Si cette partie de la liste est syntaxiquement correcte, la procédure en question renvoie à la procédure suivante un suffixe de la liste de laquelle la partie analysée a été retirée (et ainsi de suite jusqu'à obtenir un suffixe vide).

- Si la partie de la liste considérée contient des erreurs, la procédure est chargée de décrire *au mieux* chaque erreur par un message dans le fichier 'lis', et de fournir à la procédure suivante un suffixe de la liste (éventuellement la liste elle-même) dont le contenu *serait susceptible* d'être syntaxiquement correct pour les procédures suivantes.

C'est ce que nous appellerons le **rattrapage d'erreur**, qui permet de continuer l'analyse dans le but de trouver *le plus* d'erreurs *possible*.

- Si ce rattrapage s'avère impossible (par exemple parce que la liste est *trop* différente de la structure syntaxique d'une phrase), la procédure échoue, ce qui signifie l'arrêt complet de l'analyse syntaxique, et donc pas d'appel des procédures suivantes.

<u>programme</u> :

```
anal_def( [ key(DEF), mot(_name) | _reste ], _name, _reste, oui) .

anal_def( [ key(DEF), key(_key) | _reste ], _name, [ key(_key) | _reste ], non):-
                    synt_erreur_1( 'After DEF, name of object' ) .
```

```
anal_def( [ key(DEF), _found | _reste ], _name, _reste_synchro, non) :-
                synt_erreur_10( 'name of object', _found) ,
                synt_erreur_4( 'keyword : IN (if exist)' ) ,
                find_synchro( key, [ _found | _reste ], _reste_synchro) .

anal_def( [ mot(_name) | _reste ], _name, _reste_key, non) :-
                synt_erreur_10( DEF , _name ) ,
                synt_erreur_4( 'keyword : IN (if exist)' ) ,
                find_synchro( key, _reste , _reste_key ) .

anal_def( [ key(DEF) ], _ , _ , _ ) :-
                synt_erreur_15( 'DEF , and after ?' ) ,
                fail .

anal_def( [ key(_key) | _reste ], _ , [ key(_key) | _reste ], non) :-
                synt_erreur_1( 'DEF ( and the name of object associated )' ) .

anal_def( [ _found | _reste ], _ , _ , _ ) :-
                arg( 1 , _found , _f ) ,
                synt_erreur_0( _f ) ,
                fail .
```

2. <u>démonstration</u>.

La démonstration de cette procédure se réduirait à un raisonnement par cas calqué sur la postcondition de la spécification, et ne faciliterait pas plus la compréhension du problème qu'une documentation.

Nous nous contenterons donc de celle-ci.

Remarques :

- Sans le problème du *rattrapage d'erreurs* de l'analyse syntaxique, cette procédure se réduirait à sa première clause.

- Il est à noter que la cinquième clause de cette procédure n'existait pas dans le rapport de stage. La réalisation de la documentation de la procédure a permis de

vérifier si tous les cas possibles étaient traités. Il s'est avéré que non, d'où l'ajout de cette clause dont l'absence risquait d'ailleurs de provoquer le cyclage du programme.

3. documentation.

($\mu_1(m_1)$ = key(DEF) , μ_2 = mot (n >= 2))

anal_def([key(DEF) , mot(_name) / _reste], _name, _reste, oui)

(_name = m_2 , _reste = [$\mu_3(m_3)$, ... , $\mu_n(m_n)$]) .

(_bool = 'oui' , _mt = m_2 ,

$\_Lmt_2$ = [$\mu_3(m_3)$, ... , $\mu_n(m_n)$] , 'lis' = β)

($\mu_1(m_1)$ = key(DEF) , μ_2 = key (n >= 2))

anal_def([key(DEF) , key(_key) / _reste],

_name, [key(_key) / _reste], non) :-

(_key = m_2 , _reste = [$\mu_3(m_3)$, ... , $\mu_n(m_n)$])

synt_erreur_1('After DEF, name of object') { 'lis' = βM } .

(_bool = 'non' , 'lis' = β M ,

$\_Lmt_2$ = [key(m_2) , $\mu_3(m_3)$, ... , $\mu_n(m_n)$])

($\mu_1(m_1)$ = key(DEF) , μ_2 ≠ key , μ_2 ≠ mot (n >= 2))

anal_def([key(DEF) , _found / _reste], _name, _reste_synchro, non) :-

(_found = $\mu_2(m_2)$, _reste = [$\mu_3(m_3)$, ... , $\mu_n(m_n)$])

synt_erreur_10('name of object', _found) { 'lis' = βM } ,

synt_erreur_4('keyword: IN (if exist)') { 'lis' = βMM' } ,

find_synchro(key, [_found / _reste], _reste_synchro)

(_reste_synchro = [$\mu_i(m_i)$, ... , $\mu_n(m_n)$] où 2 <= i <= n,

et i est le plus petit entier tel que μ_i = 'key', s'il existe,

(sinon $\_Lmt_2$ = [])) .

(_bool = 'non' ,'lis' = β MM' , $\_Lmt_2$ = [$\mu_i(m_i)$, ... , $\mu_n(m_n)$]

où 2 <= i <= n, et i est le plus petit entier tel que μ_i = 'key', s'il existe,

(sinon $\_Lmt_2$ = []))

(μ_1 = **mot** (**n** >= **1**))

anal_def([mot(_name) / _reste], _name, _reste_key, non) :-

(_name = m_1 , _reste = [$\mu_2(m_2)$, ... , $\mu_n(m_n)$])

synt_erreur_10(DEF , _name) { 'lis' = βM } ,

synt_erreur_4('keyword: IN (if exist)') { 'lis' = βMM' } ,

find_synchro(key, _reste , _reste_key)

(_reste_key = [$\mu_i(m_i)$, ... , $\mu_n(m_n)$] où 2 <= i <= n, et i est le plus petit entier tel que μ_i = 'key', s'il existe, (sinon $\_Lmt_2$ = [])) .

(**_bool = 'non' ,'lis' = β MM' , $\_Lmt_2$ = [$\mu_i(m_i)$, ... , $\mu_n(m_n)$]**

où 2 <= i <= n, et i est le plus petit entier tel que μ_i = 'key', s'il existe,

(sinon $\_Lmt_2$ = []))

(**n = 1 et μ_1 = key et m1 = DEF**) (échoue)

anal_def([key(DEF)] , _ , _ , _) :- { }

synt_erreur_15('DEF , and after ?') { 'lis' = βM } ,

(échoue) *fail* ( ).

(**μ_1 = key (n >= 1) et m1 ≠ DEF**)

anal_def([key(_key) / _reste], _ , [key(_key) / _reste], non) :-

(_key = m_1, _reste = [$\mu_2(m_2)$, ... , $\mu_n(m_n)$])

synt_erreur_1('DEF (and the name of object associated)') { 'lis' = βM } .

(**_bool = 'non' , 'lis' = β M**

$\_Lmt_2$ = [key(m_1) , $\mu_2(m_2)$, ... , $\mu_n(m_n)$] = $\_Lmt_1$)

(**n >= 1 et μ_1 ≠ key et μ_1 ≠ mot**) (échoue)

anal_def([_found / _reste], _ , _ , _) :-

(_found = $\mu_1(m_1)$, _reste = [$\mu_2(m_2)$, ... , $\mu_n(m_n)$])

arg(1 , _found , _f) (_f = m_1),

synt_erreur_0(_f) { 'lis' = βM } ,

(échoue) *fail* ( ).

(**condition d'échec : n = 0 ou (μ_1 ≠ key et μ_1 ≠ mot**))

2.2.3. Procédures de l'analyse sémantique.

A. procédure **test_sem_cat** (page 82 du rapport de stage).

en-tête : test_sem_cat (mcat , mt)

données : mcat , mt : mots

données modifiées : bf : base de faits , 'lis' : fichier de sortie,

pré : ◇ $bf = bf_0$,

◇ 'lis', ouvert en écriture, de contenu égal à β .

post : si 'cat(mcat , mt)' $\notin$ bf ,

alors - $bf = bf_0 \cup$ 'dac(non)' ,

- 'lis' = βM où M est une suite de caractères contenant mcat et mt, et constituant un *'message d'erreur'*,

sinon ni bf ni 'lis' ne sont modifiés.

B. procédure **test_sem_attr** (page 84 du rapport de stage).

en-tête : test_sem_attr (mt , mattr)

données : mt , mattr = mots ,

données modifiées : bf : base de faits , 'lis' : fichier de sortie,

pré : ◇ $bf = bf_0$,

◇ 'lis', ouvert en écriture, de contenu égal à β .

post : si bf_0 contient un fait de la forme 'obj(prop(mt , _ , mattr , _)'

ou de la forme 'obj(fprop(mt , _ , mattr , _)',

alors - $bf = bf_0 \cup$ 'dac(non)' ,

- 'lis' = βM où M est une suite de caractères contenant mt et mattr, et constituant un *'message d'erreur'*,

sinon ni bf ni 'lis' ne sont modifiés.

C. procédure **test_sem_key** (page 83 du rapport de stage).

en-tête : test_sem_key (mcat , Lprops)

données : mcat : mot , Lpros : liste ,

données modifiées : bf : base de faits , 'lis' : fichier de sortie,

pré : ◇ $bf = bf_0$,

◇ 'lis', ouvert en écriture, de contenu égal à β .

post : si mcat ≠ key ,

alors la procédure n'a pas d'effet (ni bf ni 'lis' ne sont modifiés) ,

sinon (mcat = key)

si Lprops est unifiable avec [[[_attr] , _value]] ,

donnant _attr = m_2 et _value = L_2 ,

alors $bf = bf_0 \cup$ 'verifykey(m_2 , L_2)' ,

sinon - $bf = bf_0 \cup$ 'dac(non)' ,

- 'lis' = βM où M est un *'message d'erreur'*.

D. procédure **verif_key** (page 83 du rapport de stage).

en-tête : verif_key

données modifiées : bf : base de faits , 'lis' : fichier de sortie,

pré : ◇ $bf = bf_0$,

◇ 'lis', ouvert en écriture, de contenu égal à β ,

◇ bf_0 contient au plus un fait fact de la forme : 'verifykey(_ , L)'.

post : ◇ si fact n'existe pas, la procédure n'a pas d'effet ,

◇ si fact existe, et si L est de la forme [v_1 , ... , v_n] où n >= 1 ,

alors si ∀ i tel que 1 <= i <= n , bf_0 contient un fait de la forme :

'obj(prop(_ , part , v_i , _)' ,

alors $bf = bf_0 \setminus$ fact ,

'lis' = β ,

sinon bf = (bf_0 \ fact) ∪ 'dac(non)' ,

'lis' = βM où M est une liste de *'messages d'erreur'*,

(un pour chaque v_i ne vérifiant pas la propriété ci-dessus)

sinon (L n'est pas de la forme [v_1 , ... , v_n] où n >= 1)

- bf = (bf_0 \ fact) ∪ 'dac(non)' ,
- 'lis' = βM où M est un *'message d'erreur'*.

E. procédure test_sem_presence_cat (page 82 du rapport de stage).

en-tête : test_sem_presence_cat (mt)

données : mt : mot ,

données modifiées : bf : base de faits , 'lis' : fichier de sortie,

pré : ◇ bf = bf_0 ,

◇ 'lis', ouvert en écriture, de contenu égal à β ,

post : si mt ≠ 'entity_class' ,

ou si bf_0 contient un fait de la forme 'obj(prop(_ , part , _ , _)' ,

et un fait de la forme 'obj(prop(_ , key , _ , _)' ,

alors la procédure n'a pas d'effet (ni bf ni 'lis' ne sont modifiés) ,

sinon (mt = 'entity_class')

- bf = bf_0 ∪ 'dac(non)' ,
- 'lis' = βM où M est une liste de un ou deux *'messages d'erreur'* contenant 'key' ou (et) 'part' , selon le(s) fait(s) manquant.

F. procédure test_sem_isa (page 83 du rapport de stage).

en-tête : test_sem_isa (L , mt , mi)

données : L : liste ,

mt , mi : mots ,

données modifiées : bf : base de faits , 'lis' : fichier de sortie,

pré : ◇ bf = bf_0 ,

◇ 'lis', ouvert en écriture, de contenu égal à β ,

post : si L = [] ,

alors la procédure n'a pas d'effet (ni bf ni 'lis' ne sont modifiés) ,

sinon (L ≠ [])

- bf = bf_0 ∪ 'dac(non)' ,

- 'lis' = βM où M est une suite de caractères contenant mt et mi , et constituant un *'message d'erreur'*.

G. procédure test_sem_not_key (page 82 du rapport de stage).

en-tête : test_sem_not_key

données modifiées : bf : base de faits , 'lis' : fichier de sortie,

pré : ◇ bf = bf_0 ,

◇ 'lis', ouvert en écriture, de contenu égal à β ,

post : si bf_0 contient un fait de la forme 'obj(fprop(_ , key , _ , _)' ,

alors - bf = bf_0 ∪ 'dac(non)' ,

- 'lis' = βM où M est un *'message d'erreur'*.

sinon la procédure n'a pas d'effet (ni bf ni 'lis' ne sont modifiés) .

H. procédure flat (page 77 du rapport de stage).

(procédure qui peut échouer et dont plusieurs clauses peuvent échouer)

1. spécification.

en-tête : flat (bool , m_{name} , m_{in} , L_{isa} , L_{prop})

données : bool , m_{name} , m_{in} , L_{isa} , L_{prop}

données modifiées : bf : base de faits , 'lis' : fichier de sortie,

pré : ◇ bf = bf_0 ,

◇ 'lis', ouvert en écriture, de contenu égal à β ,

◇ bool = 'oui' ou 'non' ,

◇ m_{name} : mot ,

◇ m_{in} : mot ,

ces mots représentant un fait de la forme *'obj(in(m_{name} , m_{in}))'*

◇ L_{isa} : liste de la forme [m_1 , ... , m_n] où n >= 0 , et m_i est un mot,

cette liste représentant n faits de la forme *'obj(isa(m_{name} , m_i))'*

◇ L_{props} : liste de la forme [lps_1 , ... , lps_p]

où p >= 0 , et lps_i est une liste de la forme [mc , LP]

où mc = mot,

et LP est une liste de la forme [las_1 , ... , las_q]

où q >= 1 , et las_i est une liste de la forme [LA , v]

où v est un terme,

et LA est une liste de la forme [la_1 , ... , la_r]

où r >= 1 , et la_j est un mot .

cette liste représentant ∀ k tel que 1 <= k <= p, ∀ m tel que 1 <= m <= q,

r faits de la forme :

'obj(fp(m_{name} , mc , la_j , v))'

avec fp = *prop* si m_{in} = 'entity_class' , 'activity_class' ou 'assertion_class',

et fp = *fprop* sinon .

post : si bool = 'oui' ,

alors - la procédure réussit,

- $bf = bf_0$ ∪ l'ensemble de ces faits ∪ 'dac(non)' si les tests sémantiques ont détecté des erreurs,
- 'lis' = $\beta M_1 ... M_n$ où M_i est un *'message d'erreur'* , et n = nombre d'erreurs sémantiques contenues dans l'ensemble de faits.

sinon $bf = bf_0$ et la procédure échoue.

programme :

```
flat( non , _ , _ , _ , _ ) :-
    fail .
```

```
flat( oui , _obj_name , _obj_in , _obj_isa , _obj_properties ) :-

    metaclass( _obj_in ) ,

    assert( obj(in(_obj_name,_obj_in)) ) ,

    foreachmember( _isa_name , _obj_isa ,
        assert( obj(isa(_obj_name,_isa_name)) )
            ),

    foreachmember( [ _cat , _props ] , _obj_properties ,
        (
         test_sem_cat( _cat , _obj_in ),

         foreachmember( [ _attrs , _value ] , _props ,
              foreachmember( _attr , _attrs ,
                                  (
                                   test_sem_attr( _obj_name , _attr ),
                                   assert( obj(prop(_obj_name,_cat,_attr,_value)) )
                                  )
                           )
                       ),

         test_sem_key( _cat , _props ),
        )
            ),

    verif_key ,

    test_sem_presence_cat( _obj_in ) .


flat( oui , _obj_name , _obj_in , _obj_isa , _obj_properties ) :-

    assert( obj(in(_obj_name,_obj_in)) ) ,

    test_sem_isa( _obj_isa , _obj_name , _obj_in ) ,

    foreachmember( [ _cat , _props ] , _obj_properties ,
        foreachmember( [ _attrs , _value ] , _props ,
              foreachmember( _attr , _attrs ,
                                  (
                                   test_sem_attr( _obj_name , _attr ),
                                   assert( obj(fprop(_obj_name,_cat,_attr,_value)) )
                                  )
                           )
                       ),

    test_sem_not_key .
```

2. <u>démonstration</u>.

Etant donné la complexité de certaines **clauses** de cette procédure, nous allons décomposer cette dernière en niveaux, avec une démonstration aux niveaux inférieurs et une documentation au niveau supérieur.

a. démonstration du but, que nous appellerons B_a, de la deuxième clause:

```
foreachmember( [ _cat , _props ] , _obj_properties ,
     (
     test_sem_cat( _cat , _obj_in ),

   ┌ foreachmember( [ _attrs , _value ] , _props ,
   │     ┌ foreachmember( _attr , _attrs ,
B' │ B'' │                  (
   │     │                  test_sem_attr( _obj_name , _attr ),
   │     │                  assert( obj(prop(_obj_name,_cat,_attr,_value)) )
   │     │                  )
   │     └ )
   └ ),

     test_sem_key( _cat , _props ),
     )
          ),
```

a.1 <u>spécification</u> de B_a , B' , B" .

Ces buts sont spécifiés de la manière suivante :

{ contexte initial } B { contexte final (postcondition) }

{ C_1 : _obj_name = m_{name} , _obj_properties = L_{prop} , bf = bf_0 ,
<u>et</u> les variables _cat , _props , _attrs , _value , _attr sont non instanciées[10] }

B_a

{ bf = $bf_0 \cup$ { obj(prop(m_{name} , mc , la_j , v)) :
$\exists$ [mc , LP] $\in L_{prop}$, $\exists$ [LA , v] $\in$ LP , $\exists$ $la_j \in$ LA) }

{ C_2 : C_1 <u>et</u> _props = LP <u>et</u> _cat = mc ,
<u>et</u> les variables _attrs , _value , _attr sont non instanciées }

B'

{ bf = $bf_0 \cup$ { obj(prop(m_{name} , mc , la_j , v)) :
$\exists$ [LA , v] $\in$ LP , $\exists$ $la_j \in$ LA) }

{ C_3 : C_2 <u>et</u> _attrs = LA <u>et</u> _value = v <u>et</u> la variable _attr est non instanciée }

B"

{ bf = $bf_0 \cup$ { obj(prop(m_{name} , mc , la_j , v)) : $\exists$ $la_j \in$ LA) }

---

10. c'est-à-dire ont pour valeur elles-mêmes.

a.2 démonstration .

Dans le contexte défini ci-dessus, l'appel de B'' provoque, comme décrit dans la spécification de *foreachmember* , l'exécution de son troisième argument. Ce but est constitué de deux sous-buts regroupés entre parenthèses autant de fois qu'il y a d'éléments la_j dans la liste LA . Cette exécution réussira étant donné que tous les arguments utilisés (m_{name} , mc , la_j et v) sont bien des termes de base.

Si l'exécution du but B'' fourni les résultats correspondant à sa spécification, l'exécution du but B' se déroulera de façon similaire et réussira.

Et si l'exécution du but B' fourni également les résultats correspondant à sa spécification, le but B_a s'exécutera de manière similaire et fournira les résultats prévus dans sa spécification.

b. démonstration du but, que nous appellerons B_b , de la troisième clause:

```
foreachmember( [ _cat , _props ] , _obj_properties ,
      foreachmember( [ _attrs , _value ] , _props ,
           foreachmember( _attr , _attrs ,
                          (
                           test_sem_attr( _obj_name , _attr ),
                           assert( obj(fprop(_obj_name,_cat,_attr,_value)) )
                          )
                     )
                )
           ),
```

Cette démonstration est similaire à la précédente.

3. documentation.

(**bool** = **'non'**) (échoue)

flat(non , _ , _ , _ , _ , _) :- , { }

fail .

(**bool** = **'oui'**) (réussi si : m_{in} = 'entity_class' , 'activity_class', ou 'assertion_class')

flat(oui , _obj_name , _obj_in , _obj_isa , _obj_properties) :-

(_obj_name = m_{name} , _obj_in = m_{in} ,
_obj_isa = L_{isa} , _obj_properties = L_{prop})

(réussi si : m_{in} = 'entity_class',
'activity_class',
ou 'assertion_class')

metaclass(_obj_in) (m_{in} = 'entity_class',
'activity_class',
ou 'assertion_class') ,

assert(obj(in(_obj_name,_obj_in)))

(bf = bf_0 ∪ (obj(in(m_{name} , m_{in})) }) ,

foreachmember(_isa_name , _obj_isa ,
assert(obj(isa(_obj_name,_obj_isa)))
)

(∀ i tel que $m_i \in L_{isa}$, bf = bf_0 ∪ obj(isa(m_{name} , m_i)) }) ,

B_a

(bf = bf_0 ∪ (obj(prop(m_{name} , mc , la_j , v)) :

∃ [mc , LP] ∈ L_{prop} , ∃ [LA , v] ∈ LP , ∃ la_j ∈ LA))

verif_key (bf = bf_0 ∪ ('dac(non)') si erreur sémantique) ,

test_sem_presence_cat(_obj_in)

(bf = bf_0 ∪ ('dac(non)') si erreur sémantique) .

(**bool** = **'oui'** et $m_{in} \notin$ ('entity_class', 'activity_class', 'assertion_class'))

flat(oui , _obj_name , _obj_in , _obj_isa , _obj_properties) :-

(_obj_name = m_{name} , _obj_in = m_{in} ,
_obj_isa = L_{isa} , _obj_properties = L_{prop})

assert(obj(in(_obj_name,_obj_in))) ,

(bf = bf_0 ∪ (obj(in(m_{name} , m_{in})) }) ,

test_sem_isa(_obj_isa , _obj_name , _obj_in) ,

(bf = bf_0 ∪ ('dac(non)') si erreur sémantique) .

B_b

(bf = bf_0 ∪ (obj(fprop(m_{name} , mc , la_j , v)) :

∃ [mc , LP] ∈ L_{prop} , ∃ [LA , v] ∈ LP , ∃ la_j ∈ LA))

test_sem_not_key (bf = bf_0 ∪ ('dac(non)') si erreur sémantique) .

2.2.4. Procédures particulières.

A. procédure **lstmember** (page 48 du rapport de stage).

(procédure récursive et qui peut échouer)

1. spécification.

en-tête : lstmember (e , L)

données : e , L

pré : L est une liste.

post : si e est un élément de L,

alors la procédure réussit,

sinon elle échoue.

programme :

```
lstmember ( _X , [ _X | _Y ] ) .
lstmember ( _X , [ _T | _Q ] ) :-
        lstmember ( _X , _Q ) .
```

2. démonstration par induction sur la structure de L.

Les cas suivants sont possibles :

Cas 1. **L = [e | L']** (e est le premier élément de L)

a. L'appel de la procédure *matchera* avec la tête de la première clause car _X *matche* avec e et _Y *matche* avec L' qui est bien une liste.

b. Comme le corps de cette clause est la suite vide, l'état initial et l'état final du contexte coïncident et sont tels que _X = e et _Y = L' .

c. L'exécution de la procédure **réussit** .

Cas 2. **L = [e' | L']**

avec e' ≠ e .

a. L'appel de la procédure ne *matchera* pas avec la tête de la première clause car e ≠ e' .

Il *matchera* avec la tête de la seconde car _X *matche* avec e , et [_T | _Q] avec [e' | L'] ,donnant _T = e' et _Q = L' .

b. L'exécution du but lstmember(_X , _Q) dans ce contexte provoque l'appel de procédure :

lstmember (e , L')

qui est compatible avec la spécification puisque e est un terme de base et L' est une liste.

Par hypothèse d'induction, l'appel récursif de la procédure réussira si e est un élément de L' et échouera si e n'est pas un élément de L'.

c. Dans le premier cas, l'exécution de la procédure réussira, ce qui est correct puisque : **e ∈ L' ⇒ e ∈ L** .

Dans le second, l'appel de la procédure échouera puisque la clause considérée est la dernière de la procédure, ce qui est correct puisque :

(e ≠ e' et e ∉ L') ⇒ e ∉ L

Cas 3. **L = []**

a. L'appel de la procédure ne *matchera* pas avec la tête de la première clause car L ≠ [_X | _Y] .

L'appel de la procédure ne *matchera* pas avec la tête de la deuxième clause car L ≠ [_T | _Q] .

b. L'exécution de la procédure **échoue** .

c.q.f.d.

3. documentation.

(**L = [e | L']** où e est un terme de base et L' est une liste)

lstmember(_X , [_X / _Y]) (_X = e , _Y = L') .

(**e ∈ L**)

(**L = [e' | L']** avec e' ≠ e) (réussi si : e ∈ L')

lstmember([_X , [_T / _Q]) :-

(_X = e , _T = e' , _Q = L')

(réussi si : e ∈ L') *lstmember(_X , _Q)* (**e ∈ L'**) .

(**e ∈ L**)

(condition d'échec : **L = []** ou **L = [e' | L'] et e ∉ L'**)

B. procédure égal

(procédure qui contient des paramètres 'données modifiées' ,
et qui peut échouer)

1. spécification.

en-tête : égal (a , b)

données modifiées : a , b

pré : a = a_0 , b = b_0 où a_0, b_0 sont des termes quelconques.

post : si a_0 et b_0 sont unifiables,

alors l'appel réussit ,

et a = b = t où t est l'unificateur le plus général de a_0 , b_0 ,

sinon l'appel échoue .

programme :

égal (_X , _X) .

2. démonstration.

Démontrons qu'avec la sémantique de D.Log. telle que nous l'avons définie, l'exécution de la procédure donne bien le même résultat qu'en Prolog, c'est-à-dire donne la valeur de l'unificateur le plus général des deux paramètres, s'il existe, et échoue sinon.

L'exécution de cette procédure déclenche l'exécution de l'algorithme :

AP(égal , (a_0 ,b_0) , C)

Cet algorithme consite à rechercher la première clause telle que les paramètres de cette clause puissent être unifiés avec les arguments, c'est-à-dire dans ce cas, les paramètres de la première clause (_X , _X) avec les arguments (a , b) .

Selon la définition de l'unification, celle-ci consiste à trouver un terme t et une substitution S tels que t s'obtient aussi bien en instanciant (a , b) par S qu'en instanciant (_X , _X) par S .

La seule instanciation possible de (_X , _X) se fera par une substitution (_X = t*) , et donc le terme t sera de la forme (t* , t*) .

Si on trouve une instanciation de (a , b) égale à (t* , t*) , c'est qu'il existe une substitution telle que t* s'obtient aussi bien en instanciant a qu'en instanciant b , c'est donc que a et b sont unifiables.

Ainsi, la première et unique clause de la procédure ne *matchera* que

si a et b sont unifiables.

Et dans ce cas, l'exécution de AP(égal , (a ,b) , C) renverra comme résultat (t* , t*) .

Dans le cas contraire, la procédure échouera puisqu'il n'y a pas d'autre clause.

Tout cela correspond bien à la spécification de la procédure.

c.q.f.d.

3. documentation.

(a_0 et b_0 sont unifiables)

egal (_X , _X)

(_X = t où t est l'unificateur le plus général de a_0 , b_0) .

(a = b = t où t est l'unificateur le plus général de a_0 , b_0)

(condition d'échec : a_0 et b_0 ne sont pas unifiables)

3. COMMENTAIRES SUR LA METHODOLOGIE.

Ce chapitre expose un certain nombre de remarques issues de l'application pratique de notre méthodologie au programme de compilation.

3.1. Nécessité de spécifications précises.

Durant le stage préalable, ne disposant pas d'une grande expérience en Prolog ni d'un cadre méthodologique solide, nous avons créé bon nombre de nos procédures "par tâtonnements", réalisant souvent en parallèle le travail de spécification et de programmation.

Cette façon d'agir a eu des conséquences désagréables tant du point de vue de la forme des procédures que de leur fonctionnement.

Ainsi, on se rend compte que certaines procédures auraient pu être construites sur base de raisonnements plus simples pour un effet semblable.

D'autre part, plusieurs procédures affectaient parfois à certains de leurs paramètres des valeurs inutiles pour leur contexte. D'autres comportaient des paramètres superflus. D'autres encore présentaient des cas d'exécution non prévus.

Ces erreurs sont apparues très clairement lorsque nous avons appliqué à ce programme la méthodologie proposée[1].

Nous pensons que la méthode consistant à spécifier complètement un problème, puis à démontrer la correction de sa solution avant même de songer à l'implémenter permet d'éviter pas mal de ces erreurs. En effet, elle aide à envisager tous les cas possibles, et incite à adopter la solution la plus simple, la plus élégante et la plus maîtrisable.

---

1. Ceci explique d'ailleurs les corrections apportées à la spécification ou au texte de certaines procédures.

Et nous sommes persuadés que les efforts fournis sont largement récompensés dans la suite par la fiabilité du résultat obtenu.

3.2. Utilité des démonstrations.

Il est vrai que la méthode que nous proposons est, sous certains aspects, plutôt rébarbative. En plus des efforts qu'elle nécessite, elle amène parfois à effectuer de longues démonstrations pour des procédures apparemment très simples.

Nous croyons cependant que, d'une part, la longueur de la démonstration d'une procédure prouve que le raisonnement pour comprendre parfaitement son exécution n'est pas toujours aussi simple ; et que d'autre part, le raisonnement concernant une procédure plus complexe ne donnera pas une démonstration beaucoup plus longue et, avec l'habitude, n'exigera pas plus d'efforts.

De plus, il n'est peut-être pas nécessaire, pour certaines démonstrations, d'aller aussi loin que nous l'avons fait dans le chapitre précédent, mais nous avons jugé bon de le faire, afin d'aller jusqu'au bout dans l'application de notre méthodologie. Par exemple, certaines procédures non récursives auraient pu être seulement documentées. Nous pensons que c'est la pratique qui permet de détecter les cas où une démonstration s'impose, les cas où une documentation suffit (la démonstration ayant été réalisée implicitement), et les cas où une spécification seule est suffisante, cette dernière restant de toute façon toujours indispensable.

Enfin, remarquons que le niveau des difficultés rencontrées pour démontrer une procédure est, à notre avis, inversément proportionnel à la qualité de ses spécifications.

3.3. Lien entre la méthodologie et le langage.

Nous avions montré dans la définition du langage que le sous-ensemble de Prolog retenu était suffisamment large que pour permettre une conversion aisée des procédures utilisées dans notre application.

La définition de la méthodologie nous montre que D.Log. est également suffisamment puissant que pour permettre d'exprimer de manière concise des raisonnements complexes.

Ainsi, en ce qui concerne l'unification, on a constaté que celle-ci se ramenait souvent à un simple *pattern matching*, parce que l'on a rarement dû recourir à la notion de *données modifiées*.

D'autre part, du fait que le langage est déterministe et que les paramètres résultats ne peuvent recevoir de valeur qu'à la fin de l'exécution d'une clause, le problème de la désinstanciation de résultats ne se présente jamais, même si certains buts échouent. Ceci permet de simplifier encore les raisonnements.

3.4. Applicabilité de la méthodologie.

Comme nous l'avons montré sur quelques exemples, les procédures écrites en D.Log. se prêtent bien à l'application de la méthodologie que nous proposons. Il faut cependant signaler que le choix de ces exemples n'est pas limitatif: toutes les procédures du programme de compilation auraient pu être spécifiées et démontrées d'une façon similaire.

Et si dans certains cas la méthodologie se révélait insuffisante, de la même façon qu'il était possible d'adapter le langage, il serait également concevable que l'on puisse adapter les raisonnements dans ces cas bien particuliers.

Conclusion.

Constatant l'existence d'un certain nombre de problèmes concernant l'utilisation du langage Prolog, nous avons voulu contribuer à en résoudre quelques-uns.

Ceci nous a amené à définir complètement et précisément un langage universel constituant un sous-ensemble de Prolog, au sein duquel de nombreux problèmes solubles par Prolog peuvent, en pratique, être également résolus.

Ce langage, baptisé D.Log., se caractérise essentiellement par son déterminisme; il exploite les avantages de l'unification, et limite les inconvénients du *backtrtacking*. Son avantage décisif est d'offrir une sémantique plus accessible que celle de Prolog, autorisant une construction rigoureuse - et surtout, démontrable - des programmes, ainsi qu'une plus grande maîtrise de leur fonctionnement, indépendamment de toute référence au comportement de l'interpréteur. De plus, les programmes écrits en D.Log. peuvent être directement exécutés par l'interpréteur Prolog (moyennant des modifications syntaxiques très simples).

Ce langage une fois défini, nous avons proposé un ensemble de règles méthodologiques destinées à faciliter le processus de spécification, de démonstration et de documentation des programmes.

Ce cadre méthodologique, d'usage général, s'est révélé bien adapté à D.Log. Il a en outre permis de réaffirmer l'importance, y-compris en programmation logique, d'un effort de précision dans la spécification des programmes, ainsi que dans leur démonstration[1].

---

1. ...ou tout au moins dans la production d'"arguments convainquants" en faveur de leur correction.

Une limite importante de ce travail est que le nombre de cas pratiques sur lesquels la méthodologie proposée a été testée ne permet pas de garantir son applicabilité aisée dans tous les cas.

Une suite intéressante de ce mémoire consisterait donc à étudier l'applicabilité du langage et de la méthodologie à des cas plus diversifiés[2], et le cas échéant, à proposer des améliorations destinées à combler les lacunes éventuellement découvertes.

Nous terminerons en rappelant le fait qu'un cadre méthodologique ne peut que faciliter le travail du programmeur, sans jamais pouvoir se substituer à sa démarche intellectuelle, à son bon sens et à son expérience. Il n'y a pas de bons programmes sans bons programmeurs.

---

2. Nous pensons notamment à des programmes capables de se modifier eux-mêmes en cours de fonctionnement ou à des problèmes "essentiellement non déterministes" comme ceux dont les solutions sont définies par des règles non exclusives.

Bibliographie.

André J., Menu J., Mueller J.P. : Un exemple pédagogique pour Prolog. In : Technique et Science Informatiques, vol 3, n° 5, 1984.

BIM S.A. : BIM-PROLOG manual. Bim, Everberg, 1985. [BIMP 85].

Cherton C. : Méthodologie de la programmation. Notes personnelles, 1985.

Clark K.L., Mac Gabe F.G. : Micro Prolog : Programmer's Reference Manual. Logic Programming Associates Ltd, 1984.

Clark K.L., Mac Gabe F.G., Steel B.D. : Micro Prolog : Programming in Logic. Prentice Hall

Clark K.L., Mac Gabe F.G. : The control facilities of I.C. Prolog. In : Expert Systems in the micro-electronic Age. ED. Mitchie D., Edimburgh University Press, 1979.

Clocksin W.F., Mellish C.S. : Programming in Prolog. Springer Verlag, 1984. [CLO 84].

Colmerauer A. : Prolog, langage de l'Intelligence Artificielle. In : La Recherche, 1984, n° 158, vol 15.

Colmerauer A., Kanoui H., Van Caneghem M. : Prolog, bases théoriques et développements actuels. In : Technique et Science Informatiques, 1983, Vol. 2 n°4.

Deville Y. : A Technique for Specifying Prolog Procedures. Research paper, 1985. [DEVI 85].

Dincbas M. : The METALOG problem-solving system. In : Tarnlund (EDS) : "Proceedings of Logic Programming Workshop", Hongrie, 1980.

Gallaire H., Lasserre C. : Metalevel control for logic programs. In Clark et al.: "Logic Programming". Academic Press, 1982.

Kowalski R.A. : Algorithm = Logic + Control. In : Communications of the A.C.M., Vol 22 n°7, 1979.

Kowalski R.A. : Logic for Problem Solving. North Holland Publishing, 1979. [KOWA 79].

Kowalski R.A. : Predicate logic as programming language. In : Information Processing 1974, North Holland Publishing c°.

Le Charlier B. : Notions de Lisp. Février 1985.

Le Charlier B. : Réflexions sur le problème de la correction des programmes. Thèse de doctorat, FNDP Namur, 1985.

Le Charlier B. : Séminaire de programmation. Notes personnelles, 1984.

Leroy H. : La fiabilité des programmes. Presses Universitaires de Namur, 1975.

Leroy H. : La fiabilité des programmes. Ecole d'été de l'AFCET. Juillet 1978.

Leroy H. : Notes sur la calculabilité. Février 1984.

Lloyd J.W. : Foundations of Logic Programming. Springer Verlag, 1984.

Van Lamsweerde A. : Introduction aux systèmes experts. Notes personnelles, 1985.

Vanhentenryck P. : Logique et bases de données. Mémoire. FNDP, 1985.

Winston H., Horn B. : LISP. Addison-Wesley, 1981.

Annexes.

CML: a Conceptual Modelling Language for software

engineering

COMPILATEUR - DECOMPILATEUR

CML <---> PIF

(Prolog Internal Formalism)

Marc Derroitte

B.I.M. Research Department

Kwikstraat 4, 3078 Everberg

Avec la collaboration de

Poi Domange

Eric Meirlaen

Sous la direction de

Raf Venken

20 décembre 1985

Ce rapport d'une centaine de pages a été réalisé au B.I.M. lors d'un stage préliminaire au mémoire. Il contient une présentation détaillée des éléments suivants :

- le langag CML et sa syntaxe,
- le formalisme PIF,
- les règles de conversion de l'un à l'autre,
- le manuel d'utilisation du programme de compilation/décompilation,
- les spécifications et le texte des programmes.

Il fait l'objet d'une publication interne au B.I.M.; il n'est pas publié, et est distribué uniquement aux membres du jury.